Applying Model Checking to Agent-Based Learning Systems

Ryan F. Kirwan

7 February 2014

Submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy

School of Computing Science
College of Science and Engineering
University of Glasgow

Abstract

In this thesis we present a comprehensive approach for applying model checking to Agent-Based Learning (ABL) systems. Model checking faces a unique challenge with ABL systems, as the modelling of learning is thought to be outwith its scope. The practical work performed to model these systems is presented in the incremental stages by which it was carried out. This allows for a clearer understanding of the problems faced and of the progress made on traditional ABL system analysis. Our focus is on applying model checking to a specific type of system. It involves a biologically-inspired robot that uses Input Correlation learning to help it navigate environments. We present a highly detailed PROMELA model of this system, using embedded C code to avoid losing accuracy when modelling it. We also propose an abstraction method for this type of system: Agent-centric abstraction. Our abstraction is the main contribution of this thesis. It is defined in detail, and we provide a proof of its soundness in the form of a simulation relation. In addition to this, we use it to generate an abstract model of the system. We give a comparison between our models and traditional system analysis, specifically simulation. A strong case for using model checking to aid ABL system analysis is made by our comparison and the verification results we obtain from our models. Overall, we present a framework for analysing ABL systems that differs from the more common approach of simulation. We define this framework in detail, and provide results from practical work coupled with a discussion about drawbacks and future enhancements.

Acknowledgements

First and foremost, the biggest help throughout this research and the writing of this thesis was my supervisor Dr Alice Miller. Her guidance helped to steer this research out of treacherous waters, and her *red pen* performed lifesaving surgery on many a terminal sentence. Thank you Alice.

Another huge thanks to Dr Bernd Porr and Dr Paolo Di Prodi: the guys with the robots. They have been fantastic collaborators and provided the initial physical systems which this research is based on. Always able to answer any technical questions, and they tackled our joint work with full enthusiasm.

A big thanks also to my second supervisor Dr David Manlove for his attention to detail throughout all mini-viva hand-ins and presentations.

Thanks to Hamish Haridras. Lending his time and support with his thorough proof reading and graph *beautification* skills.

Also thanks to Dr Gethin Norman for kindly giving up his time to answer any questions I emailed him with –with an amazingly fast response time.

Special thanks to Dr Oana Andrei and Dr Iain McGinniss, my counsellor/office mates. And thanks to everyone in the department whom I've had the pleasure of meeting over the years. I have learnt something valuable from everyone. Even if it was just the positive impact of always bringing a smile to work, thanks Ittoope Puthoor.

A thanks also to the EPSRC for their generous funding of this PhD, and to the University of Glasgow staff for their help and support throughout.

Thanks to all my supportive friends, near and far. Particularly to my Ultimate Frisbee team mates. The sport has kept me fit and the friendships have picked me up on many occasions.

A final huge thanks to my family. To my wee sister Sonya, a constant source of inspiration –winning all sorts of prizes with her degrees. And especially to my Dad, a pillar of strength throughout my life. Thanks for always managing to restart my motivation by showing an unwavering interest in my research, and for running a fine-toothed comb through the entirety of the thesis.

Contents

List of Figures

Chapter 1

Introduction

In this thesis we introduce Agent-Based Learning systems (herein referred to as ABL systems). We describe a formal analysis of some example ABL systems using *model checking* combined with *abstraction*. In the context of this thesis, an ABL system contains one or more identical agents; where an agent is a system composed of both hardware and software components.

Historically, studies of ABL systems have relied on simulation; where properties are inferred from averaging results obtained by running sets of simulations. Simulation is the prevailing methodology for analysing ABL systems because it is cheap and relatively easy to do. Additionally, ABL systems are usually considered too complicated for a more formal method of analysis to be used. In this thesis we apply the formal method of model checking to ABL systems.

Model checking allows us to formally verify a system's properties. From this, definitive statements can be made as to whether a system's specification has been fulfilled. As ABL systems are complicated, it is nontrivial to apply model checking to them, and hence a sophisticated abstraction is needed.

There are several reasons for applying a more formal approach to the analysis of ABL systems; e.g., it is often unsatisfactory to rely on approximate results when systems are mission critical –or contain vulnerable/expensive components. Additionally, model checking allows us to prove properties that hold for all executions of a system –as opposed to just one execution at a time.

In this thesis we show that formal verification is a viable technique for proving properties of ABL systems pre-deployment; furthermore, that combining formal verification with simulation can lead to a greater level of confidence in the expected behaviour of a system.

Although model checking can be used as a standalone technique, we combine it with a tailor-made abstraction. Abstraction is a method for reducing the size of a model while preserving in it the properties of the original system that is being modelled. Many different abstraction approaches are available, hence identifying a suitable method for the case of ABL systems is one of our primary goals. In Chapter 5 we present a method of abstraction which we have adapted and modified for use with ABL systems. We also provide an extensive proof of the correctness of this abstraction.

In Chapter 2 we give some background to our area of research, providing a general description followed by a study of specific aspects in more detail. In Chapter 3 we describe the preliminary practical work done, which was undertaken to highlight the problems involved in modelling ABL systems.

We present our most detailed model of a specific type of ABL system in Chapter 4. In addition to our model we present our simulations of this system, and give a comparison of the results from the different approaches.

The main contribution of this research is focused on in Chapter 5, where our abstraction method and its proof of soundness are covered. Following this, in Chapter 6 we present a model that is generated from our abstraction method.

In Chapter 7 we present a comparison of the different analysis techniques for ABL systems, and describe possible extensions and improvements to our models. Lastly, we summarise our contribution and propose future work. Additional material can be found in the appendices.

Note that all the modelling and verifications we present were conducted on a 2.5Ghz dual core Pentium E5200n processor with 3.2Gb of available memory, running UBUNTU (9.04), SPIN $6.2.3$ [1],[1] and PRISM $4.0.3$ [2].

[1]The preliminary SPIN models were checked using versions from $5.2.2$ to $6.0.1$.

1.1 Thesis Statement

It is possible to aid the analysis of an ABL system by using model checking and abstraction. We create an abstraction method for ABL systems and develop standardised techniques for modelling their learning and behaviour.

1.2 Terminology

Throughout, we use the following notation.

Term	Meaning
Robot:	the physical system of an agent.
Agent:	the software representation of a robot.
Model:	the software specification of a system in a modelling language.
State-space:	the underlying set of states and transitions that are represented by a model. We use it as an alternate term to finite state machine.
Model checking:	when as a verb, to check a model for the satisfaction of logic formulas.
Property:	something that can be true or false for a given system.
Formula:	represents a test for a given property.
Verification:	the process of proving a property to be true.
Simulator:	the software specification of a system, from which simulations can be run.
Simulation:	a specific run in a simulator, representing an individual path in the system.
Explicit model:	the name of our most detailed model of a specific environment and robot.
Agent-centric abstraction:	the method we use to represent an entire class of ABL system in one model.
Relative model:	a specific PROMELA instantiation of our Agent-centric abstraction; i.e., one model for one class of system.
Cone of influence:	the area used to represent the robot and environment in a Relative model.
Polar coordinate:	(*distance*, *angle*), where *distance* is measured from a fixed point (pole), and *angle* is measured clockwise from a line projected North from that pole (polar axis).

Table 1.1: Terminology for this thesis.

1.3 Declaration of joint work

Throughout this thesis we will refer to work covered in the following joint publications: [3] and [4]. Some of the diagrams in this thesis appear in these publications. Additionally, some of the text in this thesis is an expanded version of material from these publications.

1.4 Motivation

The physical ABL systems we focus on in this work were developed in the University of Glasgow's Electronics and Electrical Engineering department (EEE). The researchers at the EEE were interested in assessing learning in biologically inspired robots. Their particular focus was on the assessment of a variety of simplistic learning algorithms, such as Temporal Difference Learning, Input Correlation Learning (ICO), and Hebbian Learning [5, 6]. Experiments involved the assessment of how well a particular robot configuration and learning algorithm fared in a given type of environment. We focused specifically on a type of system in which robots emulated primitive beetles. These robots use a dual antenna system to navigate environments.

The robots had a short pair of antennas which generated an inherent pain signal from colliding into objects, and a long pair of antennas which they learnt to use over time. The sense of pain was the stimulus for learning to utilise their long antennas in order to avoid receiving further pain signals. The particular interest of researchers at the EEE was whether the robots would be able to successfully navigate a variety of environments (without crashing) by learning to respond more (or less) vigorously to signals from their antennas. Specifically, they were interested in whether a given learning algorithm would eventually stabilise for a given system setup (as the algorithms were potentially unstable).

The general approach to the assessment of these systems was to develop a simulator and run simulations to gauge long-term behaviours. In addition, the physical systems would also be developed and tested. Our agenda was to help the EEE by providing a more formal and rigorous assessment of these systems. Particularly assessing whether robots would always eventually avoid colliding into other objects, and whether a given learning algorithm would stabilise for a specific type of robot and environment.

Chapter 2

Background

In this chapter we introduce the background material to this thesis. We give an overview of the areas involved in ABL systems and model checking. The specific ABL systems that we model are described in detail. Following this, we describe and define the mathematical constructs and techniques associated with model checking. In Section 2.4 we cover model checkers and modelling languages, particularly PRISM, PROMELA and SPIN. Then we explain a variety of techniques for abstracting systems. Lastly, we provide a detailed analysis of related literature.

2.1 Overview

A general overview of our application of model checking to ABL systems is represented in Figure 2.1, which illustrates the process of modelling a real system and proving its properties via model checking.

We start with the *Real System* which is then translated into a software program. The translation into a program is shaped by the properties of interest; i.e., we can simplify the program if we are not concerned with all the properties of the system. Hence, the translation is done in unison with selecting which of the system's properties to check. The next stage is to represent the program as a set of states and transitions, and from here we combine states or remove them via abstraction.

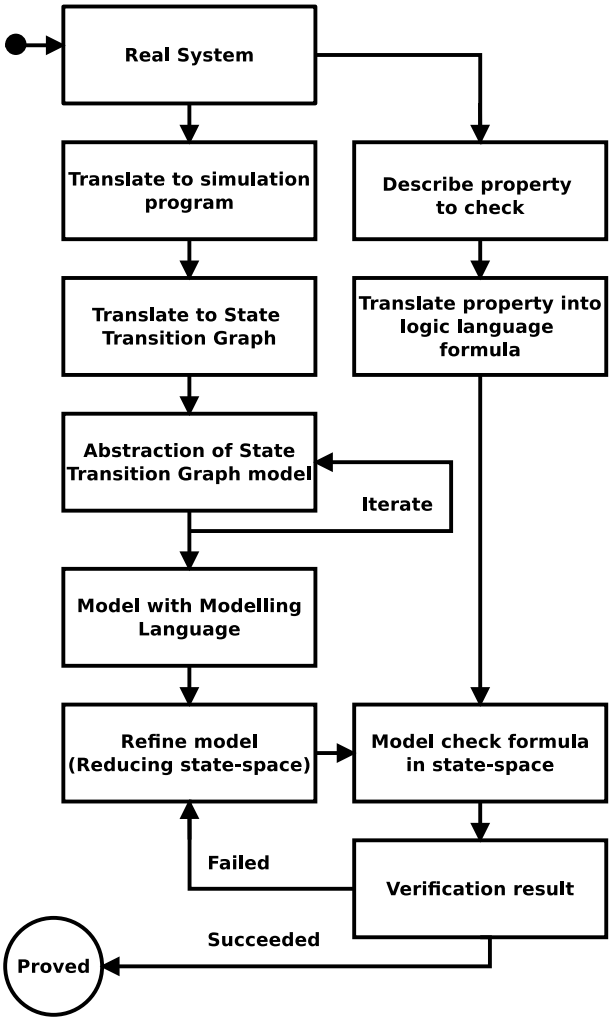


Figure 2.1: General overview of our application of model checking.

In parallel with this, the property is translated into a logic formula with a view to use it for model checking. When the state transition graph has been abstracted as far as possible, it is translated into a modelling language. Once in this form, the model is checked for the satisfaction of the logic formula. The result of a failed verification can be used to refine the model; this involves correcting inaccuracies and removing unnecessary information. In addition a failed verification may also indicate a problem with real system, or the property being checked –note that we have omitted loops that could be involved in correcting the real system, simulation program, or property. When a verification succeeds, the property is said to have

been proved.

One of the main obstacles faced when dealing with computerised systems is being able to achieve *validation of design* [7]: being able to assert whether a system will achieve its goal with a measurable degree of accuracy. This type of validation is necessary for all systems and what level of validation can be achieved is particularly relevant. We propose that model checking can provide the required level of *validation of design* for ABL systems. This is achieved by the automated verification of their properties in the formal framework of model checking.

Currently, the approach used to predict how successfully an agent in an ABL system will learn is to run many computer simulations. This process can take large periods of time and may produce an inaccurate idea of how the real system works; where the inaccuracy is due to the inability to analyse all possible simulation setups.

The inefficiency in this approach prompted our research into applying model checking to ABL systems. Having a general overview of how an ABL system behaves is not normally sufficient when developing it into a commercial system; it is more important to have guarantees that the system will never fail in a certain way, or should always, eventually reach a predefined target. These are the type of guarantees which we can provide by applying model checking.

Our research has highlighted three main difficulties when deciding how to model ABL systems; they arise from the underlying complexity of these systems and are best described as the following questions. Which modelling language and model checker should we use? How can systems be abstracted to a degree that yields a tractable state-space, while guaranteeing that the properties of the original system still hold? And, how can we accurately model and assess a learning agent? In this thesis we address these questions.

2.2 Physical systems

In this section we describe the physical hardware and underlying electronics of the ABL systems we model, beginning with a formal definition of an agent followed

by that of its components.

2.2.1 Agent Definition

We use the definition of an agent from [5]:

> "An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators."

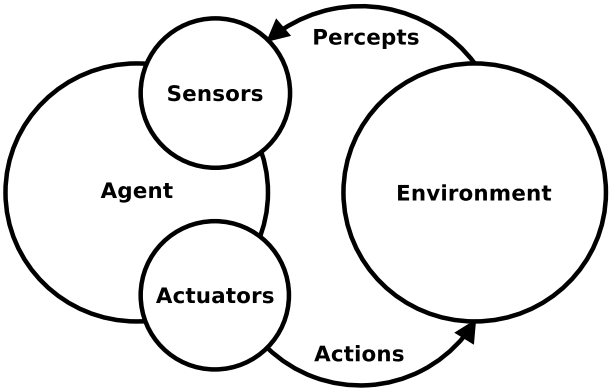


Figure 2.2: Interaction between agent and environment.

In Figure 2.2 (based on a figure from [5]) the agent perceives information (*percepts*) from its environment via sensors. It is able to perform internal calculations with the information it perceives before using its actuators to interact with its environment (*actions*).

2.2.2 Environment

We define an environment as an area in which an agent can navigate. Environments can contain obstacles, which are impassable by an agent. Environments are considered to be static areas which have no means of perceiving an agent and no means to process information.

Obstacles are considered to have a uniform size for a particular environment. There is also a minimum spacing between obstacles defined for each environment.

We refer to this distance as the *environmental complexity* and use this value to distinguish between environments. The higher the environmental complexity the smaller the minimum distance between obstacles. It is important to note that environmental complexity is not defined as a uniform distance between obstacles, only the minimum: environments can have obstacles placed at distances greater than its environmental complexity.

2.2.3 Hardware

To model ABL systems we must consider an agent's hardware components and the nature of its underlying circuitry. In the systems we model, the agents are biologically inspired robots. They are composed of actuators and sensors, as defined in Section 2.2.1. In our case, the actuators are motors designed for moving and turning, and sensors are antennas that receive percepts from the environment. The antennas are used to sense contact with another surface.

The robot uses an internal *feedback loop* in order to learn to use its sensors to activate its motors. This loop involves the robot's perceived output being fed back into the calculations for its actions. The robot is to avoid collisions by using its sensory information to guide its movement.

Percepts

Here we describe the inputs of the robot; i.e., how it uses its antenna sensors. This will provide a clearer overview of how the robot interacts with its environment.

Figure 2.3.A depicts the basic proportions of an agent in our ABL systems. Here the proximal sensors are shown to be noncontiguous with the distal sensors (unlike the situation in the real system, simulations, and models). They are shown like this to illustrate that they are distinct sensors with a shorter length than the distal sensors.

When contact is made with a proximal sensor the robot receives a signal of a preset magnitude that emulates a *painful* experience for the robot. When contact is made with the distal sensor it sends a signal to the robot of variable strength,

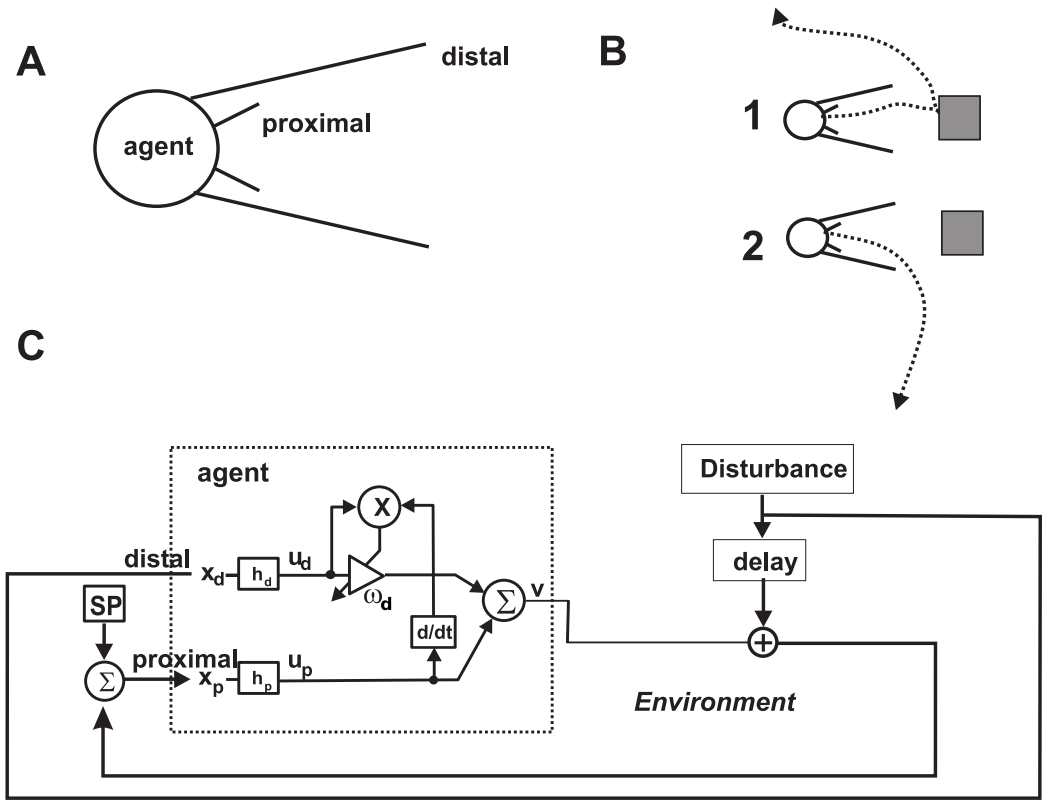


Figure 2.3: Generic closed-loop data flow with learning. A: sensor setup of the robot consisting of proximal and distal sensors. B.1: reflex behaviour; B.2: proactive behaviour. C: simplified circuit diagram of the robot and its environment (SP=set point, X is a multiplication operation changing the weight ω_d, Σ is the summation operation, d/dt the derivative, and h_p, h_d low pass filters).

where the closer to the robot that the sensor is contacted, the stronger the signal. All the signals are combined within the robot as inputs to its internal feedback loops.

The robot uses its internal feedback loops to learn to move towards or away from obstacles. This is achieved by using the difference between the signals received from its left and right pairs of antenna sensors, which can be interpreted as error signals [8]. At any time an error signal x is generated of the form:

$$x = sensors_{left} - sensors_{right} \tag{2.1}$$

where $sensors_{left}$ and $sensors_{right}$ denote the signals from the left and right pairs of sensors. The value of x is then used to generate the steering angle v, where $v = \omega_d * x$, where ω_d has a constant polarity. The polarity of ω_d determines whether the behaviour is classed as *attraction* or *avoidance* [9]. This calculation is done as part of an internal feedback loop. The loop here is established as a result of the robot responding to signals from its sensors by generating motor actions (with its actuators) which affect future signals from the robot's sensor inputs. Hence, the robot's movement influences its sensor inputs, which forms a closed loop (nominally a feedback loop, see Figure 2.3.C).

Actuators

The actuators on the robot are what it uses to affect its environment. It has motors, attached to wheels, for driving itself forward; they propel the robot in a continuous forward motion. It also has a motor for turning, which it uses to avoid obstacles. The magnitude and direction of a turn is determined by the robot's internal feedback loop.

Learning

The ABL systems we look at use various learning methods, these include: Temporal Difference Learning, Input Correlation Learning (ICO), and Hebbian Learning (see [5] and [6] for more details). Learning dynamically changes a system model and hence greatly expands the relative state-space for that model. This expansion makes verifying properties less computationally viable. In order to incorporate learning into our models we must somehow represent the process of learning within the robot.

Feedback loop In order to learn, a robot interprets the signals from its antenna sensors into its feedback loop (see Figure 2.3.C). Its actuators allow interaction with its environment and percepts provide the feedback signal. Thus, representation of the actuators and percepts is required to model the robot's learning and learned behaviour.

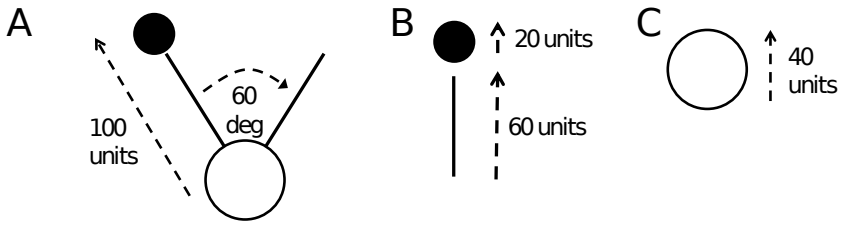


Figure 2.4: Robot setup. A: Distance from centre of robot to far edge of obstacle. B: lengths of antenna and obstacle. C: diameter of robot.

Robots in our ABL systems respond to input signals by changing trajectory to avoid collisions. Figure 2.3.B1 and B2 illustrate this response. In B1 the robot crashes into the box with its proximal antenna (after previously contacting the box with its distal antenna) and learns that this is *painful*; consequentially, in B2 it perceives the box as a potentially *painful* signal with its distal antenna and uses its actuators to change trajectory and, in doing so, avoids crashing.

This signal and response cycle forms the feedback loop for our ABL systems. It is the process by which a robot learns in these systems and is best described by ICO learning. In order to represent this accurately the details of ICO learning must be expressed in our models.

2.2.4 Input correlation learning

Input Correlation Learning (ICO) involves learning by correlating different signals from an environment in order to achieve a desired result. In the case of our ABL systems, robots try to correlate two types of signal, one from their distal antennas and the other from their proximal antennas. A robot receives a *pain* signal when it senses an impact on one of its proximal antennas. The desired result is to avoid experiencing the *pain* signal; the robot uses its distal antennas for this purpose. The proportions of the robot and its distal antennas for our ABL systems are depicted in Figure 2.4.

When a robot receives an impact on a proximal antenna it correlates this signal with the previous signal from its corresponding distal antenna. Over time the robot begins to associate between the two antennas and learns to avoid obstacles based

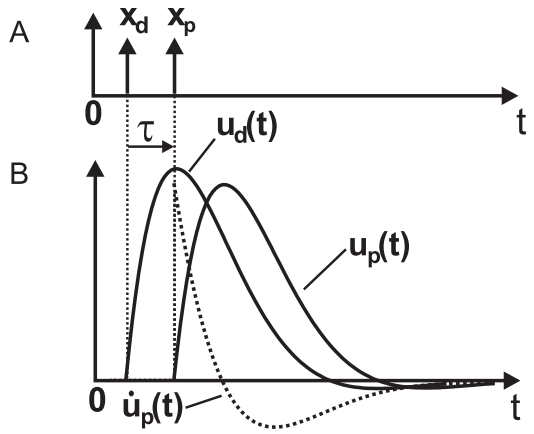


Figure 2.5: Impact signal correlation with the help of low pass filters. A: the input signals from both the distal and proximal antennas which are τ temporal units apart. B: the low pass filtered signals u_p, u_d and the derivative of the proximal signal $\dot{u}_p$.

on only the signal from its distal antennas.

On a hardware level the robot receives signals at different times, distal then proximal. Because of this time difference, correlation between the two signals is not possible. To solve this, the signals from the distal and proximal antennas are passed through low-pass filters. The filters act as a robot's short-term memory allowing it to correlate the signals.

This is illustrated in Figure 2.5 which shows the signals from the distal and proximal antennas. Signals are represented as simple pulses in Figure 2.5.A. In Figure 2.5.B the signals have been passed through a low pass filter, where they are elongated over the time axis. For a correlation to take place we use the derivative of the proximal signal $\dot{u}_p$. This means that when there is a proximal signal, $\dot{u}_p$ has a peak shifted to earlier in phase than u_p. This peak can now be correlated with the distal signal u_d. Learning stops if u_p is constant which is the case when a proximal antenna is no longer triggered. A sequence of impacts consisting of at least one impact on a distal antenna followed by an impact on a proximal antenna causes an increase in the response (by a factor λ known as the *learning rate*). See [10] for a more detaied description of ICO/differential Hebbian learning.

2.3 Model checking

Model checking is a technique in formal methods in which a brute force approach is applied to prove properties of finite-state systems [7]. The applications of model checking range from hardware component analysis to network security testing (software). If a system can be expressed as a state-space, model checking can be used to analyse it.

In our work we apply model checking to the field of ABL systems. This involves creating a finite state representation of an ABL system (i.e., a state-space); where the state-space must comprise all possible states of that system. Once modelled as a state-space it can be checked for the satisfaction of logical properties, which includes checking for: *deadlocks*, *assertions*, *non-progress cycles*, *invalid end states*, and properties expressed in *temporal logics* (see Section 2.4.1, and for temporal logics Section 2.3.10).

Our goal is to achieve *validation of design* for ABL systems; this is done by verifying all properties relevant to a system's specification. Model checking lets us check logical properties for all states that relate to the specification, and thus allows us to achieve our goal.

This section describes details of model checking, these include: the two main types of model checking that we consider, temporal logics, the different structural representations of a system as a state-space, techniques specific to the application of model checking, and techniques specific to the reduction of a model's state-space.

2.3.1 Explicit state model checking

Explicit state model checking refers to the way that the state-space is stored when checking properties. States are represented explicitly; i.e., they are not abstracted or merged. The advantage of this approach is that it is possible to trace a path from any given state back to its initial state. This allows us to provide *counterexample* paths for property violations, as opposed to simply stating that they are false. A counterexample consists of a path in which the property is violated, and

by reviewing it one can identify how the violation occurred. It does, however, require more memory to store an explicit state model than a symbolic state model; this reduces the maximum size of state-space that can be explored, compared to symbolic state model checking. The model checker SPIN [1] uses explicit state representation.

2.3.2 Symbolic state model checking

In symbolic state model checking the state-space is stored in a reduced form. The state-space is represented symbolically as a binary decision diagram (see Section 2.3.9). With this compressed representation, it is possible to check properties over much larger state-spaces than with explicit state model checking. However, this method of storage means that counterexample paths cannot be produced. This is due to the way paths are analysed when checking properties; i.e., groups of states are dealt with at once, which means that an individual path cannot be produced from verification results. The model checker PRISM [2] uses symbolic state representation.

2.3.3 Logical properties

Model checking allows for the checking of properties that are defined in a temporal logic language. One of the most important features of model checking is the ability to verify properties specified in temporal logic. In SPIN we verify properties expressed in Linear-time Temporal Logic (LTL) [11]. In PRISM we specify our properties in Probabilistic Computational Tree Logic ($PCTL$) [12].[2] These logics are described in the Section 2.3.10.

2.3.4 State-spaces

State-spaces can be represented as State Transition Graphs (STGs), where they involve a set of nodes joined by edges. In this context, a node represents a state

[2]PRISM also supports: CTL, $Probabilistic\ LTL$, CSL, and $PCTL^*$ [13]. Most of which have not been used and are omitted here.

and an edge a transition between states. These states and transitions are used to represent the workings of a system. A state is *labelled* as the set of *atomic propositions* that are true of the system in that state; where an atomic proposition is a statement concerning variables of a system, which evaluates to true or false at any state.

These visualisations of the state-space can be more formally represented in order to apply model checking. For example, they can be represented as Kripke structures (see Section 2.3.5). Once a system's state-space is expressed in this form, model checking allows for automated verification of its properties; these properties are expressed in temporal logics, e.g., LTL, CTL [14], or $PCTL$ [12]. Model checking allows properties to be tested on all paths within a state-space, where a path constitutes a set of contiguous transitions and states. This process allows us to identify states or paths that violate a given property; hence, demonstrating that the property is false. If no violating states or paths are found then the property is verified. Therefore, model checking provides a means to exhaustively test properties of our systems.

Different model checkers use different formal representations of their state-spaces; we describe some of these representations in the following sections.

2.3.5 Kripke structures

A Kripke structure is a nondeterministic finite state machine designed to represent the behaviour of a system [7]. Nodes represent different states of a system and edges represent state transitions. Each node has a *label* which corresponds to the set of atomic propositions that hold for that state. The choice of the transition at a given state is nondeterministic: no transition is more or less likely to occur than another. Temporal logics can be interpreted in terms of Kripke structures.

Figure 2.6 depicts a Kripke structure containing four states; it is represented graphically as an STG. The edges are directed, where the arrows denote the direction of the transition.

The labels for the states are: (A, B), $(\neg A, B)$, $(A, \neg B)$, and $(\neg A, \neg B)$. Label (A, B) represents atomic propositions $A \wedge B$; i.e., A and B are true in this state.

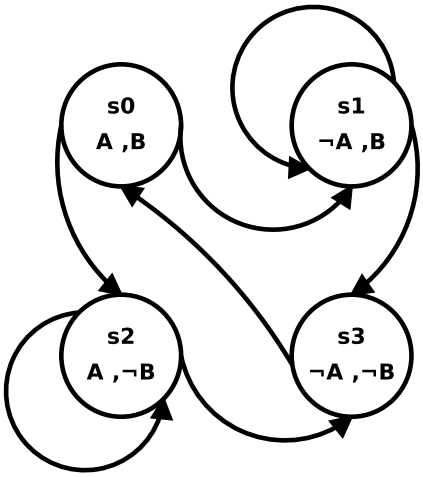


Figure 2.6: Kripke structure.

The formal definition of a Kripke structure, from [7], is as follows.

Definition 2.1. A Kripke structure is a tuple $\mathcal{M} = (S, S_0, R, L)$ where:

- S is a non-empty finite set of states.
- $S_0 \subseteq S$ is a set of initial states.
- $R \subseteq S \times S$ a transition relation.
- $L : S \rightarrow 2^{AP}$ a labelling function. For a given state s, the label for s ($L(s)$) consists of all the atomic propositions that hold in that state.

A path, π, in $\mathcal{M}$, starting at $s_0 \in S_0$ is an infinite sequence of states $\pi = s_0, s_1, s_2, \ldots$ where $\forall i > 0, (s_{i-1}, s_i) \in R$. A state $s' \in S$ is reachable in the path π if $\exists s \in S$ such that $(s, s') \in R$.

If a Kripke structure has a single initial state s_0, it is defined more simply as $\mathcal{M} = (S, s_0, R, L)$. Kripke structures differ from STGs in the way that states are labelled because Kripke structures can have labels for states that do not exist in the real system –as opposed to only those states that exist in the real system.

2.3.6 Discrete time Markov chains

Discrete Time Markov Chains (DTMCs) differ from Kripke structures in that they allow the modelling of probabilistic choices, where outcomes are partly random. From a state in a DTMC, there is a probability assigned to each transition that can occur. Diagrammatically they are synonymous with Kripke structures, but with the addition of probabilities on edges.

DTMCs are directed graphs that represent systems where transitions occur at discrete time-steps. They have a probability assigned to every transition; such that the sum of all transitions from any given node is 1. Like all of the other Markov processes considered in this chapter, they abide by the *Markov property*, which states that all transitions from a state do not depend on future or past states.

Figure 2.7 illustrates the structure of a DTMC. There are four states *s0*, *s1*, *s2*, and *s3*; arrows indicate the direction of an edge (transition). The probability of a transition is indicated by the number closest to the corresponding edge. This definition of a DTMC is based on those given in [15, 16].

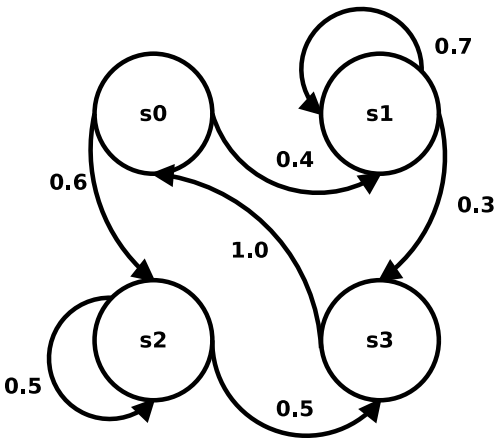


Figure 2.7: Example DTMC.

DTMCs can be used to represent systems where the likelihood of taking an action is measurable. For example, suppose there is a robot that, when turning to avoid an obstacle, three-quarters of the time will turn to its left and otherwise turn to its right. We can represent this behaviour more accurately with a DTMC than a Kripke structure. In this example, we do so by assigning a probability

of 0.75 to the transition corresponding to turning left and 0.25 to the transition corresponding to turning right.

2.3.7 Continuous time Markov chains

Continuous Time Markov Chains (CTMCs) are an extension to DTMCs where time is considered to be continuous. The time taken for an event to occur is considered to be a random variable taken from an exponential distribution. Probabilities are represented over continuous time as rates. Each transition has a rate assigned to it, which represents the probability of a transition over time. Hence, instead of a set of transitions and associated probabilities a CTMC has a *transition rate matrix* that represents the rates assigned to the transitions between states. The probability of making a transition between two states by the time t is calculated by $1 - e^{-\lambda * t}$, where λ is the average number of times that a transition is taken per unit of time t [16].

2.3.8 Markov decision processes

Markov Decision Processes (MDPs) have the same expressivity as DTMCs with the addition of being able to represent nondeterministic choices also. Hence, MDPs can represent graphs where the probability of taking a transition from one state to another is unknown. (This definition of an MDP is based on that presented in [17].)

Figure 2.8 illustrates the structure of an MDP. Each state has a set of associated actions. Edges that connect a state to an action are represented by dashed lines and denote nondeterministic choice. The actions here are *a0* and *a1*. For each action, there is a probabilistic choice of transitions (denoted by solid lines). Note that, for any actions, the probabilities of its associated transitions sum to 1.

From state *s2* there is a nondeterministic choice of selecting either action *a0* or *a1*. If *a0* is selected then there are two edges that can be chosen. One edge that represents the transition back to *s2*, and the other that represents the transition to *s1*. The corresponding probabilities are 0.6 and 0.4.

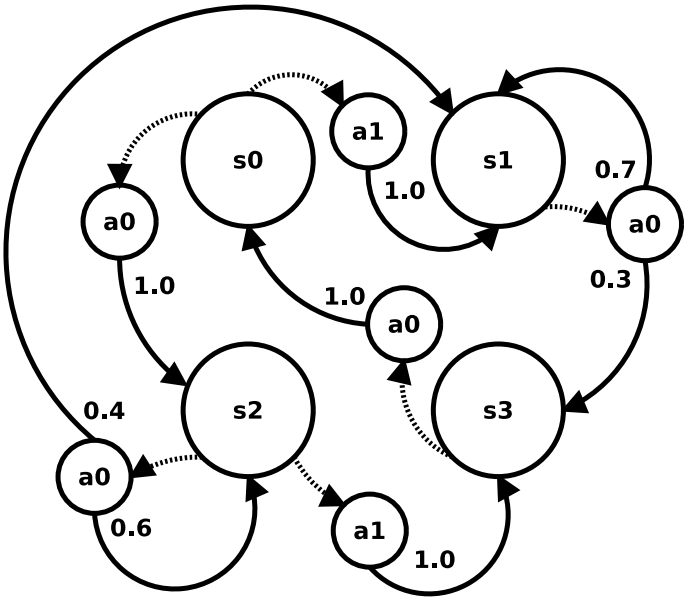


Figure 2.8: Example MDP.

MDPs are used to represent systems that contain both probabilistic and nondeterministic transitions. For example, a robot moving in an environment may have a trajectory that is determined probabilistically, yet make some choices (how to respond to an obstacle, say) nondeterministically. This type of system cannot be represented by a Kripke structure or DTMC alone.

Rewards in MDPs

MDPs may also include a reward function that is associated with each state, and a value function that calculates the measure of reward associated with a system path. Having reward functions allows the quantitative assessment of paths in the MDP. For example, suppose we have a system where a robot is navigating an environment in search of food, where the environment is composed of both food areas and obstacle areas. Then if we assign a high value of reward to a robot finding food, we can check a system for paths where the total reward reaches a predefined target –a required total amount of food to be eaten.

2.3.9 Binary decision trees/diagrams

Binary Decision Trees/Diagrams (BDTs/BDDs) are branching directed acyclic graphs used to represent boolean functions. Each node is a decision node that can have two child nodes (which are also decision nodes). Child nodes are reached by the evaluation of boolean variables, where either a 0 or a 1 is chosen for each variable at a decision node. This explanation is based on that in [7] and [18].

Table 2.1 represents a truth table for a 3-input AND function. The AND function returns *true* if all inputs are 1, otherwise it returns *false*.

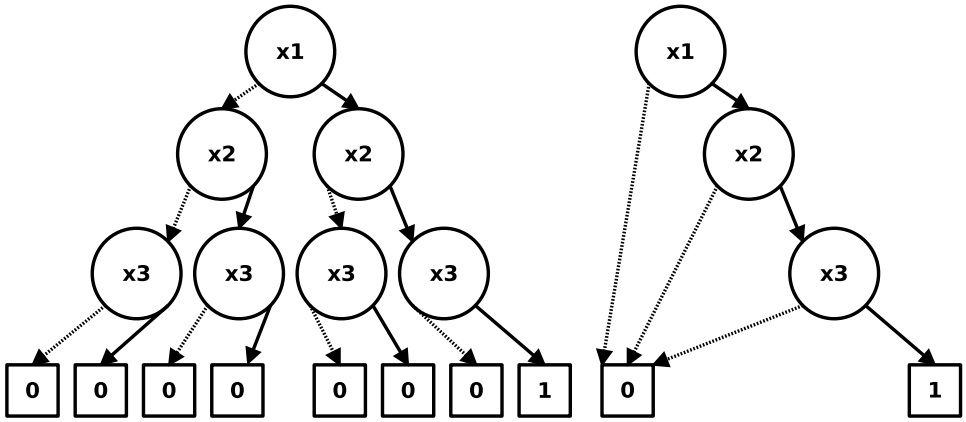
x1	x2	x3	AND
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Table 2.1: AND truth table.

Figure 2.9(a) shows the corresponding BDT. The evaluation of each boolean input variable is represented by either a dashed or solid line, corresponding to an evaluation of 0 or 1 respectively. Following a path of input values from $x1$ results in either a 0 or a 1 in the leaves at the bottom (represented as squares); a 0 indicates the function returns *false* and a 1 indicates *true*. For example, an input of 0, 0, 0 leads to the bottom left of the BDT, resulting in *false*.

Figure 2.9(b) shows the equivalent BDD. It is a more compact representation of a BDT, where decision nodes are combined. It represents the same function as the BDT; i.e., an input of 0, 0, 0 still evaluates to *false*.

The ordering of the decision nodes in a BDD can directly affect its size (although, this is not apparent in our example). BDDs are one of the data structures used for state-space storage in PRISM, where various compression techniques are applied to them to reduce their storage requirements, e.g., the use of *sparse ma-*



(a) BDT of an AND function (b) BDD of an AND function

Figure 2.9: Examples of BDT and BDD representation.

trices [16]. Other data structures used by PRISM include Multi-Terminal BDDs (MTBDDs) –defined in [19].

2.3.10 Temporal logics

Temporal logics [14] are used to define properties for model checking, and they consist of a syntax and semantics. Properties contain temporal operators that allow one to reason about the ordering of events. We are primarily concerned with *safety properties* and *liveness properties* (see Section 2.3.12). These informally have the form: something bad will never happen, and something good will eventually happen respectively.

Note that not all LTL properties can be declared as either safety or liveness. A property that falls into this category is one involving the until operator (U). For example, the property x until y ($x\,U\,y$) is both a safety and a liveness property. This property states that: x is at least true until y is true, and that y is always eventually true. The safety part of the property is associated with checking that x is never false before y; and, the liveness part with checking that y always eventually becomes true.

Syntax and semantics of CTL^*, CTL and LTL The properties that we define for our PROMELA models are written in LTL. Here we derive LTL from its superset CTL^*. We provide a formal definition of these languages as follows, where the definitions are taken from [20].

Definition 2.2. The logic CTL^* [21] is defined as a set of state formulas, where the CTL^* state and path formulas are defined inductively below. The quantifiers A and E are used to denote *for all paths*, and *for some path* respectively (where $E\phi = \neg A \neg \phi$). In addition, X, U, $<>$, and $[\,]$ represent the standard *next time*, *strong until*, *eventually* and *always* operators (where $<> \phi = true\, U \phi$ and $[\,]\phi = \neg <> \neg\phi$ respectively). Let AP be a finite set of propositions. Then:

- for all propositions $p \in AP$, p is a state formula;
- if ϕ and ψ are state formulas, then so are $\neg\phi$, $\phi \wedge \psi$, and $\phi \vee \psi$;
- if ϕ is a path formula, then $A\phi$ and $E\phi$ are state formulas;
- any state formula ϕ is also a path formula;
- if ϕ and ψ are path formulas, then so are $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, $X\phi$, $\phi U \psi$, $<> \phi$, and $[\,]\phi$.

Definition 2.3. CTL [22] is a branching time logic which expands the state-space in a tree structure. It allows for quantification over paths, but cannot describe individual paths –like LTL. The logic CTL is the sublogic of CTL^*; where its temporal operators X, U, $<>$, and $[\,]$ must be immediately preceded by a path quantifier.

Definition 2.4. The logic LTL [11] is also a subset of CTL^*. It is obtained by restricting the set of CTL^* formulas to those of the form $A\phi$, where ϕ does not contain A or E. When referring to an LTL formula, the A operator is generally omitted and instead the formula ϕ is interpreted as "for all paths ϕ".

Definition 2.5. For a model $\mathcal{M}$, if the CTL^* formula ϕ holds at a state $s \in S$ then we write $\mathcal{M}, s \models \phi$ (or simply $s \models \phi$ when the identity of the model is clear from the context). The "models" relation $\models$ is defined inductively below. Note that for a path $\pi = s_0, s_1, \ldots$, starting at s_0, $first(\pi) = s_0$ and, for all $i \geq 0$, π_i is the suffix of π starting from state s_i. Then:

- $s \models p$, for $p \in AP$ if and only if $p \in L(s)$;
- $s \models \neg\phi$ if and only if $s \not\models \phi$;
- $s \models \phi \wedge \psi$ if and only if $s \models \phi$ and $s \models \psi$;
- $s \models \phi \vee \psi$ if and only if $s \models \phi$ or $s \models \psi$;
- $s \models A\phi$ if and only if $\pi \models \phi$ for every path π starting at s;
- $\pi \models \phi$, for any state formula ϕ, if and only if $first(\pi) \models \phi$;
- $\pi \models \neg\phi$ if and only if $\pi \not\models \phi$;
- $\pi \models \phi \wedge \psi$ if and only if $\pi \models \phi$ and $\phi \models \psi$;
- $\pi \models \phi \vee \psi$ if and only if $\pi \models \phi$ or $\pi \models \psi$;
- $\pi \models \phi U \psi$ if and only if, for some $i \geq 0$, $\pi_i \models \psi$ and $\pi_j \models \phi$ for all $0 \leq j < i$ (note that if $i = 0$ then the property is trivially true, and there is no need to evaluate j);
- $\pi \models X\phi$ if and only if $\pi_1 \models \phi$;
- $\pi \models <> \phi$ if and only if $\pi_i \models \phi$, for some $i \geq 0$;
- $\pi \models [\,]\phi$ if and only if $\pi_i \models \phi$, for all $i \geq 0$.

Syntax and semantics of $Probabilistic\ CTL$ **(**$PCTL$**)** We use the temporal logic of $PCTL$ [12] to describe properties in our PRISM models. $PCTL$ allows us to express properties to do with probabilistic models; i.e., models with probabilities associated with transitions. The following definitions are taken from [23] (for a more explicit definition of $PCTL$ see [24]).

Definition 2.6. For a state s, $Path_s^{fin}$ and $Path_s$ denote the sets of all finite and infinite paths starting from s. In order to quantify the probability that a DTMC satisfies a given property, we define, for each state $s \in S$, a probability measure $Prob_s$ over a $Path_s$.

- For a finite path $\pi \in Path_s^{fin}$, the probability of path π occurring is $P_s(\pi_{fin})$. The i^{th} state of path π is denoted $\pi(i)$.
- Let n be the number of states in the path π such that $n = |\pi_{fin}|$.
- If $n = 0$, then $P_s(\pi_{fin}) = 1$;
- otherwise $P_s(\pi_{fin}) = [\ P(\pi(0)) \times P(\pi(1)) \times ...P(\pi(n))\ ]$.

There is also need to define the *cylinder set* $C(\pi_{fin})$.

- $C(\pi_{fin})$ is the set of all infinite paths π that occur after π_{fin}.

Definition 2.7. The set of $PCTL$ state and path formulas are defined inductively over a finite set of propositions, over system variables. The probability measure $Prob_s$ is unique such that $Prob_s C(\pi_{fin}) = P_s(\pi_{fin})$, when $\forall\ \pi_{fin} \in Path_s^{fin}$. There is also a *bounded until* operator $\mathbf{U^{\leq k}}$.

- $\pi_1 \mathbf{U^{\leq k}} \pi_\mathbf{2}$ is $true$ if $\pi_1 \mathbf{U} \pi_\mathbf{2}$ is $true$ and π_2 is satisfied within k time steps.

We also define how a state s satisfies a property.

- Let $\bowtie$ be a relation where $\bowtie \in \{\leq, <, >, \geq\}$.
- Let $\mathbf{P}$ be a probability.
- Let ψ be a path property.

State s satisfies $\mathbf{P}_{\bowtie \mathbf{p}}[\psi]$ if the probability of taking a path from s that satisfies ψ is within the parameters of the relation $\bowtie$.

- State formulas include $true$, $false$, $(v_i = d_i)$ and $(v_i \neq d_i)$. Also, if ϕ and ψ are state formulas, then so are $\neg\phi$, $\phi \wedge \psi$ and $\phi \vee \psi$.
- If ϕ is a path formula, then $\mathbf{P}_{\bowtie \mathbf{p}}[\phi]$ is a state formula for any $\bowtie \in \{\leq, <, >, \geq\}$. It is also the case that any state formula ϕ is also a path formula.
- Other path formulas are $\mathbf{X}\phi$, $\phi \mathbf{U} \psi$ and $\phi \mathbf{U^{\leq k}} \psi$, provided that ϕ and ψ are state formulas

Definition 2.8. $PCTL$ logic is the set of all state formulas. For a DTMC, *D*, if the $PCTL$ formula ϕ holds at a state $s \in S$ then this can be shortened to: *D*, $s \models \phi$. If the formula ϕ does not hold then we write *D*, $s \not\models \phi$. For a path formula ψ and a state s, $P_s(\psi)$ is defined as $Prob_s(\{\pi \in Path_s : \pi \models \psi\})$ where $Prob_s$ is the probability measure on $\sum_s$, as defined in Definition 2.7. The relation, $\models$, is inductively defined below.

- $s \models true$, and $s \not\models false$.
- $s \models (v_i = d_i)$, if and only if $s = (e_1, e_2, \ldots e_k)$ and $e_i = d_i$.
- $s \models (v_i \neq d_i)$, if and only if $s = (e_1, e_2, \ldots e_k)$ and $e_i \neq d_i$.
- $s \models \neg\phi$ if and only if $s \not\models \phi$.
- $s \models \phi \wedge \psi$ if and only if $s \models \phi$ and $s \models \psi$.
- $s \models \phi \vee \psi$ if and only if $s \models \phi$ or $s \models \psi$.
- $s \models \mathbf{P}_{\bowtie \mathbf{p}}[\phi]$ if and only if $P_s(\phi) \bowtie p$.

- $\pi \models \mathbf{X}\phi$ if and only if $\pi(1) \models \phi$.
- $\pi \models \phi \mathbf{U}^{\leq \mathbf{k}} \psi$ if and only if for some $i \leq k$, $\pi(i) \models \psi$ and $\pi_j \models \phi \; \forall \; 0 \leq j < i$.
- $\pi \models \phi \mathbf{U} \psi$ if and only if for some $k \geq 0$, $\pi \models \phi \mathbf{U}^{\leq \mathbf{k}} \psi$.

Given $D = (S, s_0, P)$, if π_s is a path starting from any state $s \in S$, then we say that D, $\pi_s \models \phi$ if and only if $D_{\mathbf{s}}$, $\pi_s \models \phi$, where $D_{\mathbf{s}} = (S, s, P)$.

2.3.11 Büchi automata and LTL

One of the most efficient algorithms for model checking LTL properties is the automata-theoretic approach [25]. Although we will not describe the algorithms in detail, we provide a little background theory here.

Definition 2.9. A state-space $\mathcal{A}$ is a tuple $\mathcal{A} = (S, s_0, L, T, F)$ where:
1. S is a non-empty, finite set of states
2. $s_0 \in S$ is an initial state
3. L is a finite set of labels (on transitions)
4. $T \subseteq S \times L \times S$ is a set of transitions, and
5. $F \subseteq S$ is a set of final states.

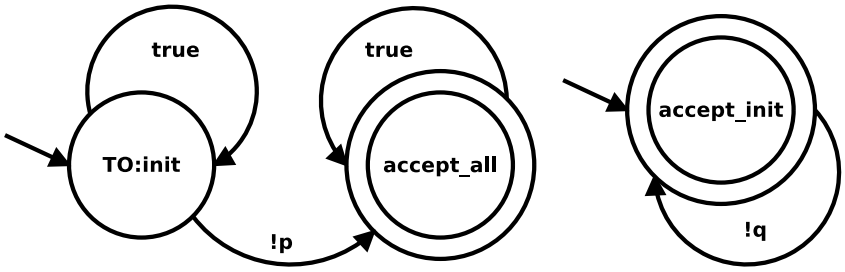
A run of $\mathcal{A}$ is an ordered, possibly infinite, sequence of transitions

$$(s_0, l_0, s_1), (s_1, l_1, s_2), \ldots$$

where $s_i \in S$ and $l_i \in L$ for all $i > 0$. An accepting run of $\mathcal{A}$ is a finite run in which the final transition (s_{n-1}, l_{n-1}, s_n) has the property that $s_n \in F$.

In order to reason about infinite runs of an automaton, alternative notions of acceptance are required; e.g., Büchi acceptance. We say that an infinite run (of a state-space) is an accepting ω-run (i.e., it satisfies Büchi acceptance) if and only if some state in F is visited infinitely often in the run. A Büchi automaton is a state-space defined over infinite runs (together with the associated notion of Büchi acceptance).

Every LTL formula can be represented as a Büchi automaton (see, for example [26] and [27], and references therein).



(a) $\neg[\,]\, p$ (b) $\neg <> q$

Figure 2.10: Example Büchi automata

In Figure 2.10 we give Büchi automata for the negation of the LTL formulas $[\,]p$ (p is true at every state) and $<> q$ (q is true eventually). Note that any path of an automaton A has associated paths in the Büchi automaton. For example, consider the Büchi automaton of Figure 2.10 (a). If π is a path in A for which p becomes false at some state s say, it would be possible to loop around the state labelled `T0:init` until s is reached, then make a transition to the (acceptance) state labelled `accept_all`. The infinite continuation of π would result in infinite looping around the acceptance state in the Büchi automaton. Thus π would be *accepted*. Similarly, a path π' in A for which q is never true would be accepted by the Büchi automaton of Figure 2.10 (b).

Note that the state labels of Figure 2.10 are not significant, apart from the fact that the acceptance states are prefixed with the term `accept`. Büchi automata are used when model checking with SPIN, where they are represented as *never claims* (see Section 2.4.1). The labels used here are those generated for the corresponding never claims by SPIN.

2.3.12 Searching a state-space

When model checking, the verification of properties is achieved by searching a model's state-space. Here, we describe two general types of property and one of the most common algorithms used to check them. We do this with reference to SPIN, as this is the most relevant context for our work (we justify our choice of model checker in Section 2.4.4).

Safety properties

A safety property is verified in SPIN if, when model checking, there is no counterexample path produced. Specifically, all paths within a model's state-space are checked for a violation of the safety property.

An example of a safety property is: "the red button is never pressed". To check this property SPIN searches for a counterexample where the "red button" is pressed. If SPIN cannot find this, then the property is verified. Otherwise, SPIN produces a finite counterexample path in the model which includes a state that violates this property.

Liveness properties

A liveness property is verified in a different manner than a safety. Although, it still involves searching for a counterexample, and if this cannot be produced then the property is verified. For this check, SPIN searches the model's state-space for infinite paths (cycles), which violate the liveness property.

An example of a liveness property is: "the red button is always eventually pressed". In this case, a counterexample is a path where the "red button" is never pressed. Once again, if no counterexample can be produced then the property is verified. To disprove a liveness property requires the demonstration of an infinite path in which the goal property does not hold. For this example, an infinite path constitutes looping behaviour (includes a cycle), where the model is able to go through this cycle infinitely often without the button being pressed. Hence, SPIN identifies a counterexample of this property as one that contains this violating cycle.

Depth-first search

When exploring a state-space, one of the common algorithms used is Depth-First Search (DFS). It is a simple algorithm for traversing a state-space, and can be used to check an LTL formula on-the-fly. The algorithm involves searching one path through a state-space at a time. When exploring a path, every new state

```
//Stores states in current path (ordered set).
Stack D = {}
//Stores all states visited so far (unordered set).
Statespace V = {}

Start()
{
  Add_Statespace(V, A.s0)
  Push_Stack(D, A.s0)
  Search()
}


Search()
{
  s = Top_Stack(D)

  for each (s, l, s') that is in A.T
    if In_Statespace(V, s') == false
    {
      Add_Statespace(V, s')
      Push_Stack(D, s')
      Search()
    }

  Pop_Stack(D)
}
```

Figure 2.11: Basic DFS algorithm.

encountered is stored in a state vector and every path is stored on a stack. The stack represents the sequence of states that compose the current path being explored.

The basic algorithm for DFS is shown in Figure 2.11. It is described with reference to a Büchi automaton, `A`.

DFS can be used to check for safety properties by adding a check for the property into the `Search` function. This check is applied to every state that is explored. If the property is found to be false for any state, then the contents of the stack can be used to create a counterexample (a series of states that lead to the violation).

An extension to the DFS algorithm is used to check liveness properties, namely Nested DFS (NDFS). NDFS performs a similar search as DFS, but with the purpose of exposing infinite cycles in the state-space. When checking for a liveness property, if a path is found with a state which violates that property, then NDFS searches for a transition back to a state that has already been visited on this path.

If such a state is found, this implies that there is an infinite cycle that violates the liveness property.

2.3.13 State-space explosion

State-space explosion is a common problem in the field of model checking, and concerns the rapid exponential growth of a model as its number of variables or processes increase. Therefore, as a system becomes more complex (requiring more variables to represent it) the state-space of its model will become too large to check. To overcome this limitation, techniques have been developed to reduce the impact of state-space explosion. One such technique is to use compression when model checking. Additionally, there are the more advanced techniques of partial order reduction and symmetry reduction. We describe these three techniques here.[3]

Compression

When exploring a state-space it is common to try to reduce the memory requirements by applying a form of compression. The hash-table storage structure is commonly used as a basic form of compression for storing states, where the use of a hash function prevents multiple storage of the same state. Two main categories of compression techniques are *lossless* and *lossy*. Lossless compression reduces the memory requirements by increasing the run-time when performing an exhaustive verification. Lossy also reduces the memory required for a verification, but by using approximation techniques. However, unlike lossless, it cannot guarantee that the search of the state-space is exhaustive.

A form of lossless compression used in SPIN is *Collapse* compression. It involves collapsing the global state of the system into separate components and then storing combinations of those components into larger vectors called Global State Descriptors (GSDs). Local variables (smaller state components) are stored separately from the global data objects, and assigned small unique index numbers.

[3]Note that these techniques are not restricted to SPIN.

The index numbers are then combined with the global data object in the GSDs. This reduces the amount of memory for each global state, but the extra indexing increases the run-time.

A form of lossy compression is *Bitstate-hashing* (also used by SPIN). The hash function here uses a *checksum polynomial*. Using this type of hash function can result in hash-crashing; i.e., in the case of model checking, a situation where different states are incorrectly stored in the same slot of a hash-table. This can lead to inaccurate verification results, as the state-space may not be fully explored.

Partial order reduction

Partial Order Reduction (POR) [1, 28] is a technique whereby an equivalence is determined between paths. Rather than searching the whole state-space, a reduced number of *representative* paths are explored. This results in a reduction in the number of paths followed when model checking, and often resulting in fewer states. This is done in such a way as to ensure that, if a property being checked does not hold for all paths, at least one error path is exposed. Equivalent paths are identified and represented by *equivalence classes*.

The basic premise of POR is that, in some cases, the specific ordering of certain transitions in a path does not affect the truth, or otherwise, of a property. POR can be applied if the interleaving of different transitions does not affect the property that is to be checked.

POR is available with SPIN, and is applied in an on-the-fly manner by default. The equivalence used in this case is ϕ-stutter equivalence, where ϕ is an LTL formula.[4] To ensure that a suitable set of representative paths are explored, SPIN selects a subset of transitions from any reached state s (*ample(*s*)*) rather than the entire set of enabled transitions from s (*enabled(*s*)*). This is done in a fairly conservative way, often with the result that ample(s) = enabled(s). However, the selection process is safe –no error paths are missed. Details of the selection process can be found in [1].

[4] Another equivalence used in POR is *trace equivalence* [29]. This is not used in SPIN and so we do not discuss it here.

SPIN determines ϕ-stutter equivalence by identifying a set of operations that do not affect the property to be checked and are independent of the operations available at the same state. We illustrate this concept in Figure 2.12. Here ϕ denotes the formula `[](x < 2)` (i.e., `x` is always less than 2). Note that ϕ cannot involve the "next time" operator X.

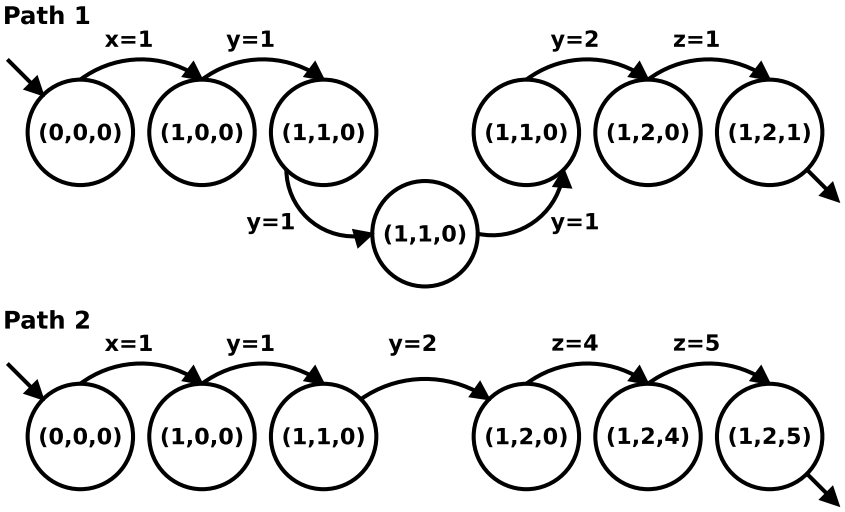


Figure 2.12: Example of POR. Each state is labelled with the values of the tuple `(x,y,z)`. The transitions are represented by arrows with their associated operation beside them. Paths 1 and 2 are ϕ-stutter equivalent.

Figure 2.12 illustrates two paths that are ϕ-stutter equivalent, with respect to property ϕ. Path 1 goes through an additional transitions (which involves a loop back to the same state: `(1,1,0)`) compared to Path 2 which includes some different states (with different values of `z`). However, property ϕ only concerns `x`, and `x` is consistent between the paths. Therefore, with respect to ϕ the paths are ϕ-stutter equivalent. This effectively means that SPIN can just explore one of these paths when verifying property ϕ, instead of both.

Symmetry reduction

Symmetry reduction [30, 31] involves exploiting symmetry between states in order to reduce the size of a state-space. Symmetry between states is determined by identifying a symmetry group of a state-space's underlying Kripke structure.

Symmetry reduction also exploits equivalence, but this time between states. Rather than storing all explored states, only certain representative states are stored. The underlying symmetry must either be provided to the model checker explicitly (by the modeller) [20], or can be inferred from the underlying symmetry of the system topology (e.g., using TopSpin [30]).

Although we do not apply symmetry reduction during model checking, we do apply it to our models explicitly in their definition. This is done by identifying and then merging states that are equivalent with the property of concern. For example, suppose we are modelling a robot and a food source, where the robot can contact the food source at three distinct points. We can reduce the state-space of this model if we are only interested in whether the robot makes contact with the food source and not the exact point of contact. Here, we can merge three distinct states, with different contact points, as they are equivalent for the purpose of our model. A more detailed explanation of how we apply this technique is given in Chapter 5.

2.4 Model checkers and modelling languages

In this section we cover several modelling languages and model checkers that were considered for the purpose of modelling ABL systems. First, we describe in detail the PROMELA modelling language, its associated model checker SPIN, and the embedded C code that can be utilised in PROMELA. Then we describe the PRISM language and model checker, and lastly we cover several hybrid modelling languages and their associated model checkers. We conclude with a comparison and summary of the languages and checkers discussed.

2.4.1 PROMELA and SPIN

Here we cover the model specification language PROMELA and the associated model checker SPIN [1]. PROMELA is a highly expressive language, and is a natural language for representing reactive systems –such as our ABL systems.

PROMELA

PROMELA provides a comprehensive set of programming constructs that enable models to be developed quickly and intuitively. Table 2.2 describes the datatypes available in PROMELA, and Figure 2.14 presents a simple sample of PROMELA code. The sample code involves two robots passing messages to one another; where one robot is sending signals, and the other is receiving signals and performing a drilling operation. In the subsequent sections we describe the main constructs of PROMELA, with reference to this example in Figure 2.14.

Type	Values	Size (bits)
`bit, bool`	`0, 1, false, true`	1
`byte`	`0...255`	8
`short`	`-32768...32767`	16
`int`	$-2^{31}...2^{31}-1$	32
`unsigned`	$0...2^n$	≤ 32

Table 2.2: Numerical datatypes in Promela.

Typedefs A *typedef* is a user-defined construct, and it can be assigned any of the legal datatypes. Typedefs provide a useful means of grouping data for message transfer via channels, and for defining multi-dimensional arrays. Figure 2.13 shows how to use a typedef to represent a coordinate.

```
typedef coord {int x; int y};
```

Figure 2.13: `typedef` example.

Proctypes A *proctype* is a template for a type of process and consists of a set of statements. Different parameter values are used to instantiate multiple processes of that type (e.g., client and server processes).

In Figure 2.14 there are three processes declared, which are: `signalRobot`, `boringRobot`, and `init`. The `init` process is used to initiate variables and run proctypes –here it runs `boringRobot`.

```
mtype = {yes, no};
chan bore = [1] of {mtype};
byte drill;

active proctype signalRobot()
{
here:  if
       :: (drill==0) -> bore!yes; goto here;
       :: (drill==1) -> bore!no; goto here;
       fi;
}

proctype boringRobot()
{
  byte power = 1;
  do
  :: ((power==1)&&(drill==0)) -> bore?yes; drill = 1;
  :: ((power==1)&&(drill==1)) -> bore?no; drill = 0;
  :: (power==0) -> drill = 0;
  od;
  }

init
{
  drill = 0;
  run {boringRobot();}
}
```

Figure 2.14: PROMELA code Boring example.

```
active proctype signalRobot()
{...}
```

Figure 2.15: `proctype` example.

Notice that in Figure 2.15 the `signalRobot` has an associated `active` keyword, this indicates that it should run as soon as model checking begins. The use of the `active` label means that a process does not need to be initiated via the `init` process. Within the Boring example the `signalRobot` process is used to send messages to the `boringRobot` process, telling it to bore if it is not and to stop if it already is.

If statements An `if` statement contains one or more lines between the reserved words "`if`" and "`fi`". Each line begins with "`::`" and represents an option in the statement. A line can start with a guard (test) which if passed the subsequent commands are performed.

```
here:  if
       :: (drill==0) -> bore!yes; goto here;
       :: (drill==1) -> bore!no; goto here;
       fi;
```

Figure 2.16: `if` statement example.

In Figure 2.16 there are two options, their guards are passed if `drill==0` or `drill==1`. For this statement we have added a label "`here`" and the command "`goto`" such that after an options' commands are evaluated the proctype (containing the `if` statement) loops back to the beginning of the statement.

Do loops A `do` loop is similar to the `if` statement. The options are contained between "`do`" and "`od`".

In Figure 2.17 there are three options, their guards relate the values of variables `power` and `drill`. Unlike the `if` statement, once a set of commands are evaluated the proctype loops back to the beginning of the statement without the need for a label.

```
do
:: ((power==1)&&(drill==0)) -> bore?yes; drill = 1;
:: ((power==1)&&(drill==1)) -> bore?no; drill = 0;
:: (power==0) -> drill = 0;
od;
```

Figure 2.17: `do` loop example.

Note that when evaluating an `if` statement or `do` loop, if no guards are fulfilled the proctype in which it is contained waits at the beginning of the statement. If all guards are fulfilled then one option is chosen nondeterministically and its commands are evaluated.

Channels A *channel* provides a means of information transfer between processes. Channels are FIFO queues, and messages are written to and read form channels. Various operations can be performed on a channel, such as random and non-destructive reads. The queue length of a channel can be set to zero, which allows it to be used as a semaphore or rendezvous channel. This type of channel can be used as a lock for shared resources.

```
chan bore = [1] of {mtype};
```

Figure 2.18: `chan` example.

In Figure 2.18 the channel `bore` is declared. It has a length of 1, and in the Boring example it is used to send "`yes`" and "`no`" messages from the `signalRobot` to the `boringRobot`. The statement "`bore!`" indicates a write operation (of the message "yes" or "no"), and "`bore?`" a read operation (the read operation is *destructive* –i.e., it removes the message from the channel).

Atomic statements *Atomic statements* provide a user-implemented means of reducing the number of interleaved paths when checking a model's state-space. This is achieved by combining statements, so that their associated transitions are forced to execute consecutively. This is effective, provided that the sequence of transitions cannot *block* in the middle, causing atomicity to be broken. The advantages of this are best described in the following example.

Suppose we have an agent in a grid, which can move between grid cells and scan the grid cell it is in. The agent's movement is controlled by one process and its scanning by another. Consider the situation when the agent moves diagonally in the grid, from (x,y) to ($x + 1$, $y + 1$). SPIN interprets this by incrementing each coordinate separately, creating an intermediate state with a coordinate ($x + 1$, y) or (x, $y + 1$), depending on the order of commands. If the coordinates' updates are not combined in an atomic statement then the robot's scanning process can interrupt the its movement process, causing additional paths to be created. This is illustrated in Figure 2.19 (*Original*), where a scan takes place at a grid cell (1,0) that the agent should not be on. Using an atomic statement means that the two coordinate updates occur consecutively, i.e. are not interrupted. Hence, the scanning process waits until the agent has completed its movement, resulting in a reduction in the number of paths. This reduction is shown by the comparison between the *Original* and *Atomic* STGs.

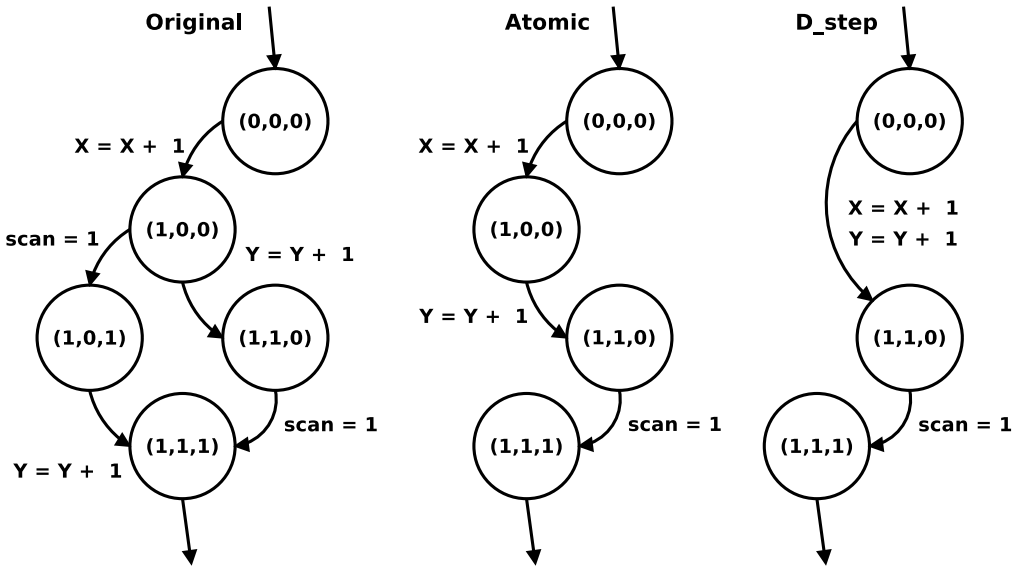


Figure 2.19: Advantages of atomic and d_step statements. The state labels are of the form (x,y,$scan$).

D_step statements A *d_step* statement is a more restrictive version of an atomic statement because it does not allow any nondeterminism, blocking, or jumping

```
inline SYMBOLIC_NAME(parameters) {
  //Segment of code.
}
```

Figure 2.20: `inline` example.

to or from the instructions it contains. Additionally, rather than just reducing the number of paths, it actually stops extra states from being created. In Figure 2.19, the *D_step* graph shows the combining of coordinates' updates to result in the removal of the extra state, (1,0,0). That is, the update of both x and y coordinates are forced to occur in one transition.

Inline functions An inline function in PROMELA is similar to a C macro. The body of an inline function replaces its *symbolic name* in the PROMELA code. The inline function can take in parameters, but does not have a return value; it may, however, be used to change the value of any variable it refers to. The format of an inline function is shown in Figure 2.20.

Never claim

LTL properties that are to be checked for in SPIN are defined in terms of PROMELA, within a construct known as a *never claim*. A never claim can be thought of as a PROMELA encoding of a Büchi automaton (see Section 2.3.11) representing the negation of the property to be checked. A never claim for the property $[]p$, corresponding to the Büchi automaton in Figure 2.10 is given in Figure 2.21.

SPIN

SPIN is an explicit-state model checker used to verify properties for models specified in PROMELA. It allows properties to be expressed as LTL formulas, which it can then verify automatically.

In this section we cover some of the main features of SPIN. First, we describe how SPIN checks a property using a never claim. Next we cover the in-built

```
 /*
 * Formula As Typed: []  p
 * The Never Claim Below Corresponds
 * To The Negated Formula !([]  p)
 * (formalising violations of the original)
 */

never  {    /* !([]  p) */
T0_init:
if
:: (! ((p))) -> goto accept_all
:: (1) -> goto T0_init
fi;
accept_all:
skip
}
```

Figure 2.21: Never claim for property $[]p$

property checks provided in its verification process. We also describe message sequence charts, which allow the flow of information in its models to be observed. Lastly, we describe SPIN's application of *fairness* on a verification: *weak fairness*. This is a way of limiting the types of path that could be deemed as error paths.

Never claim checking

SPIN creates a state-space for each process defined in a PROMELA specification. It then constructs the asynchronous product, A, of these automata and a Büchi automaton $\neg B$ corresponding to any never claim defined. The automata A and $\neg B$ are then executed in alternate steps –the propositions in $\neg B$ being evaluated with respect to the current values of the variables in A. Automaton A can be thought of as a graph in which the nodes are states of the system and in which there is an edge between nodes s_1 and s_2. If at state s_1, some process can execute a statement (make a transition) then this results in an update from state s_1 to state s_2.

Deadlock checking One of the basic checks done via model checking with SPIN is the check for a reachable system deadlock. This is an in-built check in SPIN,

```
active proctype Blender() {

end:      do
          :: (on == 1)      -> foodToLiquid = 1;
          :: (broken == 1) -> assert(on == 0);
progress:                     maintenance = 1;
                              broken = 0;
          od;
}
```

Figure 2.22: PROMELA code Blender example.

automatically applied when running a verification. A system deadlock can occur when one or more processes fail to make progress in an asynchronous model.

For example, a mutual exclusion scenario is a situation where checking for a system deadlock is valuable. Assume we have two processes, A and B, and one shared resource, X. Only one process can access X at a given time. Verifying that there is no deadlock here ensures that each process has terminated in a valid end state; i.e., they are not in an undesired state, potentially waiting to access the resource X indefinitely.

Assertion checking Assertion checking in SPIN provides the user with a means to check a global invariant. An assert statement is embedded within the declaration of a process and evaluated whenever the corresponding transition is executed. If the expression is false, an error is reported. If there is no error message, then the assertion is true.

Figure 2.22 is a sample of PROMELA code, representing the behaviour of a blender. The behaviour of the blender is contained within the proctype statement. It can be on or off, broken or fixed, and may be undergoing a maintenance operation or not, where the maintenance of the blender involves a human. It seems appropriate that the blender should be off while a maintenance operation is going on. To test this, we use an assertion to check whether the blender is off before maintenance takes place. To do this we place the statement `assert(on == 0);` within the `Blender` process as illustrated in Figure 2.22. Now, when we run a verification on this model, SPIN will produce an error if our assertion is

violated (is found to be false).

Non-progress checking In addition to adding assertions to PROMELA, one is able to reference lines of code with labels. One such label is a `progress` label, which is used in the verification process to ensure that a model will always make *progress*. What constitutes progress is decided by what the label refers to. If there are potentially infinite execution cycles in the verification process, we may want to be assured that, during these cycles, the model continues to make progress in a certain way. To do this we can add a progress label to a line of PROMELA that we want the system to continuously pass through. Then by adjusting the verification process to check for non-progress cycles (an option when running a verification with SPIN), if there are cycles that do not pass through the progress label infinitely often SPIN will report an error message.

In Figure 2.22 we have placed a progress label beside the assignment: `maintenance = 1`. This application of the label means that all cycles must pass through the maintenance operation infinitely often. Hence if the blender process is able to run infinitely often, then we can be confident that it will continue to be maintained.

Invalid end-state checking Another label in the verification process is the `end` label. It is used when one wants to identify sections of PROMELA code at which a verification can terminate, when checking properties. By default, when checking a property with SPIN the only valid end points are those where all instantiated processes have reached the end of their code, which may not reflect the actual end point of the model.

In Figure 2.22 an `end` label is placed beside the `do` loop, which means that the process can legally terminate while still in the `do` loop; i.e., during verification no error will be given if at the end of the model's execution sequence the blender process has not terminated. This seems like a reasonable end point for this model because the blender can now be left in a state where it is awaiting use.

Message sequence charts Message Sequence Charts (MSCs) are a graphical representation of communications within a SPIN model. Vertical lines represent processes and horizontal/diagonal arrowed lines express information transfer between processes, where information transfer involves message passing between channels. Boxes are used to represent message passing, where a box is shown on a vertical line that represents the process which is involved in the information transfer. Each box contains the name of the process that is using the channel and the *time-step* at which a message pass occurs (within a path of the model). Time-steps represent uniform amounts of time, where each time-step is a discrete point in time in a model. The time-step in each box is the total number of time-steps that have passed since the instantiation of the model. Figure 2.23 illustrates one example; we give other examples of MSCs associated with the SPIN practical work presented in Section 3.1.

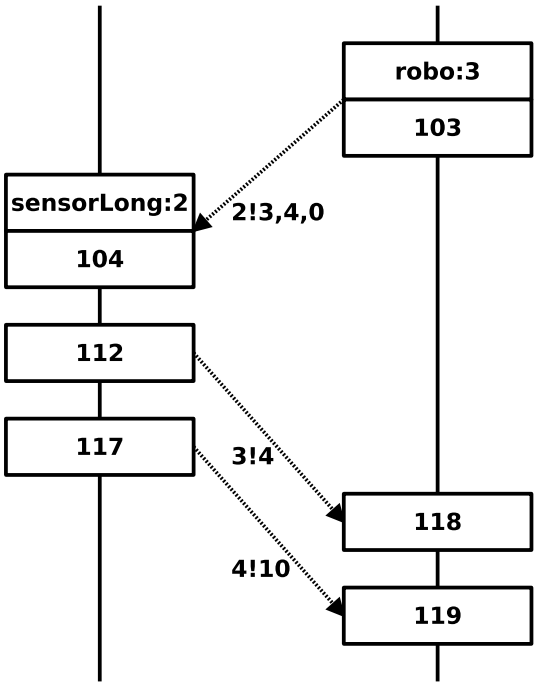


Figure 2.23: Example MSC.

Figure 2.23 depicts an agent ($robo$) sending a message to its sensor ($sensorLong$) via a channel. The first message sent is represented by "$2!3, 4, 0$", where '2' is the

index for the receiving end of the channel, and the '!' indicates that a message has been sent. The numbers following '!' denote the message. An arrow from one box that makes contact with another box indicates that its associated message has been received.

MSCs allow counterexamples (error paths) in a model to be explored graphically. This can aid in the identification of errors within the model. This analysis also means that the exact point at which a property is violated can be identified.

Weak fairness SPIN implements the feature of *weak fairness*, which allows greater user control on what constitutes a counterexample during verification. Weak fairness is applied when verifying an LTL formula, and allows us to prevent SPIN from exploring *unfair* paths. Weak fairness ensures that any process that has a continuously enabled transition, will eventually execute that transition. It allows us to avoid pathological paths in which a property fails to hold, due to a process *unfairly* failing to gain control of execution. Figure 2.24 illustrates a potentially infinite cycle which, without weak fairness, would violate the LTL formula: $[]\ <>\ (x == 4)$. Note that the label on each state indicates the value of `x`.

In Figure 2.24, the use of weak fairness forces the continuously enabled transition from state `(3)` to `(4)` to be executed.

Embedded C code

A useful feature of SPIN is that it allows for the use of `C` code embedded within a PROMELA specification as `C` code macros. The primary reason for this is to provide support for programs already written in `C` code with minimal translation into PROMELA [1], not for use in hand-written PROMELA specifications. However, in our case, the increased accuracy afforded by the use of mathematical functions available using `C` code outweighs the increased complexity resulting from its use.

Embedded `C` code allows one to reduce a model's state-space by changing variables from PROMELA into `C` code, where the variables no longer create new states (although it is possible to include them as state variables if necessary). For

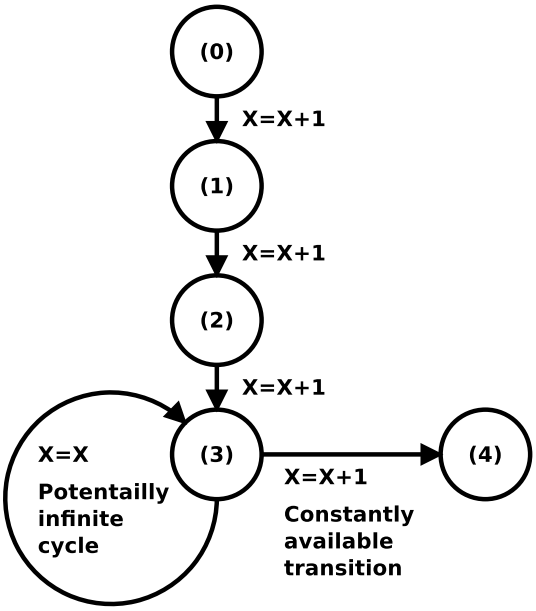


Figure 2.24: Example of *weak fairness*.

example, variables used for intermediate calculations that are not relevant (i.e., do not constitute a state vector) can be stored and manipulated in C code. This gives a significant advantage in terms of the tractability of a state-space by reducing the number of states.

In this section we describe the semantics of the embedded C code.

c_decl A c_decl primitive can appear only in the global declarations of a PROMELA specification. It allows for C code datatypes to be embedded into a model. Example 2.25 shows how a c_decl primitive is inserted into a PROMELA program.

```
c decl {
       typedef struct polarCoord {
               int d,a;
       } polarCoord;
}
```

Figure 2.25: c_decl example

Here the C code typedef is used to create a struct that is a polar co-

ordinate (containing a distance and an angle). This coordinate can be used in calculations within the PROMELA or in other sections of embedded C code, and it can be referenced in LTL formulas when checking properties of the model.

c_state A c_state primitive can appear only as a global declaration. It allows for C code variables to be defined in the PROMELA code that are part of the model's state-space. The c_state variables can be defined as *Global*, *Local*, or *Hidden*. Global variables have a common value for all processes; additionally, they are used when generating the state-space of the model. Hidden variables are declared globally in the PROMELA code although they are not used when generating the model's state-space. They can, however, be used for calculations in the PROMELA code. Local variables are local to a process, but will still be part of the state-space. In Example 2.26 three declarations of c_state variables are given.

```
c state ``polarCoord obPos''     ``Global''
c state ``polarCoord agentPos'' ``Hidden''
c state ``polarCoord antenPos'' ``Local proc1'' ``now.agentPos''
```

Figure 2.26: c_state example

There are three possible fields for each declaration, they are: variable type and name, visibility, and initial value. Note that the local variable (antenPos) has the name of the process it is local to after the *Key*word Local. Also note, antenPos is initiated to the same value as agentPos.

c_code A c_code primitive can appear anywhere within a PROMELA specification. Once reached, the C code within the c_code primitive is executed unconditionally and automatically. Example 2.27 shows how it is used.

```
do
:: c code {position1.d++; now.agentPos.d++;}
od;
```

Figure 2.27: c_code example

Here the C code is simply incrementing two integers in structs. The "now." prefix indicates that it is referring to the internal state vector for the model. (The

internal state vector is used to generate the state-space for the model.)

c_expr A c_expr primitive is equivalent to the c_code primitive except that it is not executed unconditionally. It is only executed if it returns a non-zero value when its test is evaluated. Example 2.28 presents a conditional statement that could be used in a c_expr. If the positions are equal (a non-zero value is returned), the c_code primitive is executed.

```
do
:: c expr {position1.d == position2.d}
           c code {now.agentCrash = true;}
od;
```

Figure 2.28: c_expr example

c_track Any C code variables that directly affect the value of PROMELA variables must be tracked during a verification. The c_track primitive allows us to do this. Each c_track declaration refers to the memory location and size of a C variable to be tracked. The use of this primitive allows the associated variables to be tracked during the verification of the model, while allowing for the normal verification of properties. It is important to note that even if an embedded C code variable does not directly affect a PROMELA variable, it may affect it indirectly so will still need to be tracked. A c_track declaration is shown in Example 2.29.

```
c track ``&moveDist'' ``sizeof(int)''
```

Figure 2.29: c_track example

The C code variable moveDist is tracked using the c_track primitive. Here '&' denotes the memory location, while "sizeof{int}" indicates that the variable is the size of an integer.

2.4.2 PRISM

PRISM is both a modelling language and a model checker [2], and we describe each of them in this section. We use PRISM for some of our preliminary models in

Section 3.2, and we highlight our reasons for using PRISM in Section 2.4.4.

PRISM: language

The PRISM modelling language [13] allows the inclusion of probabilistic transitions within a model; it can be used to represent DTMCs, MDPs, and CTMCs. (see Sections 2.3.6, 2.3.8, and 2.3.7 respectively). The basic datatypes of the PRISM modelling language are shown in Table 2.3.

Type	Values	Size (bits)
`bool`	`false, true`	1
`int`	$-2^{31}...2^{31}-1$	32
`double`	$-2^{63}...2^{63}-1$	64

Table 2.3: Numerical datatypes in PRISM.

We will now describe the main constructs of PRISM.

Modules A *module* contains a number of local variables. At any time, the values of these variables represent the state of that module. The global state of a model is represented by the states of all its modules. Modules also contain commands, which describe the behaviour of each module.

Commands *Commands* are the executable statements in modules. They comprise a label (name), a guard statement, and one or more variable updates. A guard statement must be satisfied before the corresponding update(s) can be executed; where each update has a given probability, and the probabilities within a command sum to 1. Example 2.30 shows a command in PRISM.

```
[example] (test=0) -> 1.0 : (object'=1);
```

Figure 2.30: Guard example

Here, the name of the command is `example` and the parenthetic statement, between the name and the arrow, (`test=0`), is the guard. The guard must be

satisfied before the update executes. The number following the arrow is the probability of the update being executed. Once the guard is satisfied, the probability of the update executing is 1.0. The update changes the next value of the variable `object` (`object'`) to 1.

Formulas *Formulas* provide a shorthand way of writing PRISM code. They consist of a unique name (identifier) and an expression; wherever the name is, the expression will be used in its place. Example 2.31 shows a formula in PRISM.

```
formula f1 = min( 10, max( x, y));
```

Figure 2.31: Formula example

Formula $f1$ finds the maximum of x and y, then chooses the minimum of that and 10. Example 2.32 shows how $f1$ can be used.

PRISM: model checker

PRISM is a symbolic-state model checker that is used to check probabilistic properties and perform quantitative analysis [14]. In PRISM, properties can be expressed in $PCTL$. The logic $PCTL$ is given in full in Section 2.3.10. Here we demonstrate the structure of common $PCTL$ properties, as expressed in PRISM.

P operator The P operator is used in $PCTL$ properties; it is used to check the probability of an event's occurrence. The following examples of the P operator highlight two ways in which it can be used.

Example 2.33 states that the probability of `pathProperty` being satisfied is greater than 0.5.

Example 2.34 is a query, expressing the question: what is probability that the *pathProperty* is satisfied by the paths from the initial state.

```
[ ] (f1 = 10) -> (x' = 0);
```

Figure 2.32: Formula example

```
P > 0.5 [pathProperty]
```

Figure 2.33: P operator: property example

```
P = ? [pathProperty]
```

Figure 2.34: P operator: query example

S operator The S operator is used to reason about the steady-state behaviour of a model. This refers to the behaviour of a model in the *long-run* or in *equilibrium*. To assess this, all states are generated and the probability of being in any given state, at any time-step, in an infinite run is calculated. The following examples of the S operator highlight two ways in which it can be used.

```
S > 0.5 [pathProperty]
```

Figure 2.35: S operator: property example

Example 2.35 states that when a model is in equilibrium, it is in a state where the probability of `pathProperty` being satisfied is greater than 0.5.

```
S = ? [pathProperty]
```

Figure 2.36: S operator: query example

Example 2.36 is a query expressing the question: what is the probability that when the model is in equilibrium it is in a state where `pathProperty` is satisfied. Note that in all these examples the property (`pathProperty`) refers to some checkable test; e.g., it could be `x = 0`.

2.4.3 Hybrid model checkers and modelling languages

In this section we describe a number of hybrid model checkers.

Overview

Hybrid models represent systems that exhibit behaviour which is subject to both discrete and continuous change. Discrete change is associated with the software of a system, and continuous change with the hardware. Hence, hybrid systems

combine software and hardware components; e.g., they have both a processor and mechanical elements. Hybrid modelling languages and model checkers have been developed to represent this type of system.

Hybrid model checkers provide a customised means of modelling hybrid systems by allowing for a reduction in a model's state-space by bounding variables within states. This bounding can be customised to restrict variables to a set range of values; (i.e., a form of data abstraction where this range of values compose a single state in the model). There are many hybrid modelling languages and model checkers that apply this form of data abstraction, and here we discuss some of the more established ones.

HyTech

HyTech is the name of both a symbolic-state model checker and its modelling language [32]. It uses hybrid automata to model systems, specifically linear hybrid automata, which have transitions for capturing discrete change and differential equations for capturing continuous change.

Linear hybrid automata are a subclass of hybrid automata that can be analysed automatically by computing with polyhedral state sets (a set of states that can be reached via one another, as time elapses).

HyTech provides diagnostic and debugging information as well as the standard true or false verifications. It provides time-stamped events which lead to error paths in verification.

Examples of modelling real systems are described in [33]. A case study is presented in which a fire brigade robot is modelled as an STG (see Section 2.3.4). HyTech allows for the discrete transitions of this STG to become continuous; hence, generating a more accurate representation of a real system. The properties of interest, in this case, relate to questions such as: do the robots try and put out the fire without having any water?

This case study also covers the concept of synchronisation in the real system and how this can be represented in the model. For example, suppose a robot is listening when a civilian cries for help. In this situation, the listening robot always

hears the civilian as soon as they cry; hence, the synchronisation of these actions should be assumed by the model. By synchronising these actions the model's state-space can be reduced when checking properties. This is an abstraction of the real system because it assumes that the two separate actions are synchronised and combines them as one action. Although this abstraction seems reasonable, it poses a problem because it means that the model is no longer an exact representation of the real system. In [33], the issues with this type of abstraction (based on synchronisation) are referred to as the *instantaneous transition problem*.

Instantaneous transition problem In traditional state transition systems it is assumed that moving from one state to another is instantaneous. For real systems, such as ABL systems, this is not the case. For example, in ABL systems agents cannot respond infinitely fast within their environment. To combat this issue the concept of *synchronisation points* is introduced in [33]. These allow the coordinated treatment of a common resource; e.g., a position in an environment, to be treated as a common resource. Hence, if multiple agents want to move to the same position, their actions would have to be dealt with by each agent sending a request to synchronise with the common resource. By having to synchronise before moving, the agents' actions are resolved in a way that better represents the real system –as opposed to agents instantly appearing at a position, or instantly crashing into one another.

We use a similar concept to synchronisation points in our practical work (see Section 3.1.1), to more accurately model our ABL systems. Other useful papers describing systems that have been modelled in HyTech, and which include tutorials that introduce the language are [34] and [35].

Alternative hybrid model checkers

A similar hybrid model checker to HyTech is the Polyhedral Hybrid Automaton Verifier (PHAVer). It has some improvements over HyTech, which are discussed in [36]. PHAVer is a tool for verifying safety properties of linear hybrid automata, providing *infinite precision arithmetic* in a robust implementation. Infinite pre-

cision arithmetic refers to calculations that are limited in precision by memory constraints. In [36] a comparison of PHAVer and other hybrid model checkers is given; where PHAVer is shown to outperform HyTech and others, including CheckMate [37] and HSolver [38].

The HySAT modelling language takes advantage of SAT solvers by integrating an arithmetic constraint solver with a tightly bounded model checker (iSAT) [39]. It uses bounded model checking and temporal formulas in order to obtain propositional SAT (boolean or propositional satisfiability) problems, on which it uses SAT solvers to perform verification.

2.4.4 Comparison of model checkers and their languages for ABL systems

It could be that there are no appropriate languages to model a system, in which case a new language may need to be created; or, it may be an option to represent a system in multiple languages to model it fully. This multi-language representation can be done by automatically converting from one language to another. In [40] one such automatic language converter is discussed.

For our ABL systems, we primarily use PROMELA and SPIN. PROMELA provides an expressive language and SPIN an established model checker with many inbuilt state-space reduction techniques (see Section 2.3.13). It is also important to note that none of the hybrid model checkers looked at here seemed to provide significant advantages that would make them worth using in preference to SPIN (for modelling our ABL systems): where it is already possible to use bounded variables or even unseen variables within embedded C code sections of the PROMELA.

Although we focus on using SPIN, we also use PRISM to generate some probabilistic models of our ABL systems. The use of a probabilistic modelling language allows for a larger variety of properties to be checked and provides a means to represent the likelihood of any transition being taken in our system models. For these models, transitions can be assigned a precise probability of being taken. PRISM also allows us to quantify the results from our verifications such that we can de-

termine the precise probability of a formula being fulfilled. Our PRISM models are described in Section 3.2.

Deriving the probabilities from our ABL systems, however, is imprecise. For example, suppose a robot is trying to turn 90° clockwise. The probability assigned to this action affects the accuracy of all properties, yet quantifying the precise probability of an exact 90° turn is not feasible. Hence, some actions are better represented without an associated probability. Additionally, PRISM does not have en equivalent to PROMELA's embedded C code, which makes in unable to express our systems with the same accuracy. Therefore, we primarily model our ABL systems using PROMELA, and check their properties using SPIN.

2.5 Abstraction

In model checking, abstraction refers to the simplification of a system's specification in order to produce a model with a tractable state-space. An abstracted model, or abstract model, should still include all states and transitions that concern properties that require checking. The intention is that from checking properties of an abstract model one can infer that they hold for the original system.

When modelling a system, its properties are expressed using a set of *Atomic Propositions* (AP); where each atomic proposition in AP evaluates to either true or false. An alternative to the 2-value evaluation of an atomic proposition, is the 3-value abstraction technique. Here atomic proposition can be assigned the value of *unknown*, which gives a formal representation of neither true nor false. The 3-value abstraction technique is used in [41, 42], and abstraction specifically for hardware verification in [43].

Abstraction involves the simplification of a specification which, consequentially, could add errors to the results of checking properties. Therefore, techniques for bounding these errors have been developed. They are used to quantify the likelihood of properties that are true for abstract models being true for their original systems. In [44] the modelling of a probabilistic system is presented, where a coffee-delivering robot system is represented as an MDP (see Section 2.3.8)

and used to explain abstraction and error bounding techniques. Additionally, a methodology for generating abstract models automatically from an MDP representation is presented.

The technique proposed for bounding errors in [44] is as follows. First, choose a set of *immediately relevant* atoms (IR). Set IR is comprised of the atoms which have the greatest impact on the reward function of the model. Next, a set of *relevant* atoms (R) is chosen, where R is defined as the smallest set of atoms that satisfy the following rules: R includes at least all the atoms in IR; and if there is an atom x which is in set R and is the result of an action A on another atom y, then y is also in set R.

Our systems are abstracted based on the properties that we wish to check, and we do not use rewards. It is conceivable to incorporate rewards into our models, as the choice of property to check implies a notion of reward. Hence, we apply the same methodology, but without the MDP formalism of rewards.

In [45] a method of approximating to the minimal abstraction is presented. This is referred to as inexact abstraction. In this case the abstraction can be derived directly from the text of the program without having to construct the original transition system. The approach of inexact abstraction is *conservative*, which means that it is restricted to properties with the *for all* ($\forall$) path qualifier of the form $\forall\ CTL$ (see Section 2.3.10). Therefore, if a property is *true* for the abstracted model then it is also *true* for the original system. However, if a property is shown to be *false* no conclusion can be drawn with respect to original system.

The technique of counterexample abstraction is presented in [46]. It uses an automatic iterative abstraction methodology, which involves analysing the control structures within a program to create an initial abstract model. States in the abstract model are generated by clustering states from the real system. This produces an abstract model based on a set of AP. The labelling for a cluster of states is the same as for all of the individual states in the cluster.

A model is checked for properties that fulfill the system's specification, where a counterexample demonstrates an error. If these properties are shown not to hold for the abstract model then they are referred to as *spurious* counterexamples.

These are used to highlight states and transitions that cause the abstract model to be inaccurate. The states are then removed from the abstraction, or simply altered enough to make the property hold. Iterating this process is used to refine the abstract model to make it more concise and accurate.

2.6 Autonomous agents and multi-agent systems

In this section we discuss literature specific to the area of multi-agent systems (herein referred to as MA systems).[5] Most of the papers here appear in the Autonomous Agents and Multi-Agent Systems journals.

2.6.1 Representing MA Systems

There are many difficulties when modelling systems that comprise autonomous agents. One of the main difficulties is how to represent an autonomous agent in a modelling language. Modelling different types of agents can be time consuming when there is no common language to describe an agent's behaviour. In this situation each agent is analysed individually, which can lead to no obvious commonality between agents and no reusable description of an agent. In light of this, common languages have been developed called *agent-oriented* languages, such as 2APL [48], 3APL [49], and AgentSpeak [50]. AgentSpeak is a logic-based programming language centred on the Beliefs, Desires, and Intentions (BDI) architecture.

The BDI architecture represents an agent as balancing its time equally between "choosing what to do" and "doing it". The time taken for an agent to plan ahead (think) –as opposed to just choosing an action at a given time– is not within the scope of the BDI architecture.

BDI represents three aspects of an agent, they are its: state represented by its beliefs, goals represented by its desires, and intentions represented by its plans to

[5] MA systems overlaps with some ABL systems, but they do not necessarily involve a learning element. A broad introduction to MA systems is given in [47].

achieve the goals. It provides a software model for intelligent agents that separates them into the three discrete elements of beliefs, desires, and intentions.

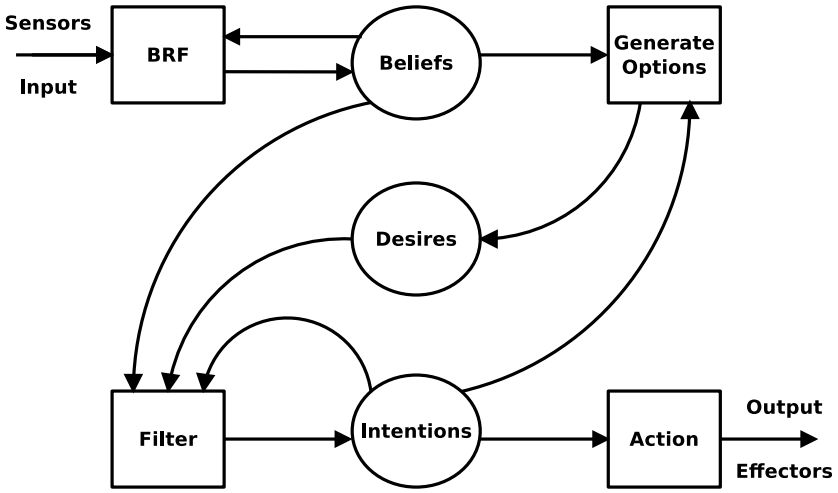


Figure 2.37: Generic BDI architecture.

Figure 2.37 is based on a figure from [51], and represents the generic architecture of a BDI agent. It shows how the agent receives information from its percepts (sensors) into the *belief revision function* (BRF). The BRF receives the combination of information from an agent's percepts in order to generate new *beliefs*; its formation of new beliefs is also affected by the agent's current beliefs.

Beliefs are what the agent *thinks* about the state of its environment. Based on these beliefs, the agent generates options that can be chosen from its current state. The number of options generated is affected by the *intention* of the agent, where the agent's intention is the current plan it is following. Options are evaluated with the *desires* of the agent –the agent's goals. The *filter* generates new intentions based on beliefs, options, and previous intentions. Having the intentions in a feedback-loop with the rest of the model allows the agent to form new plans from those it is already committed to.

2.6.2 Formal approaches

This section focuses on attempts to formally represent and analyse MA systems.

In [52] a formal development framework for describing an MA system is presented. Three attributes which are common to the logic used to represent MA systems are highlighted, they are: an *informational component*, which represents an agent's beliefs or knowledge; a *dynamic component*, which represents dynamic activity in the system; and a *motivational component*, which represents an agent's desires, intentions, and goals. The framework only focuses on temporal logic and it has been developed for the specification, verification, and implementation of MA systems. However, the framework specifically focuses on the areas of process control, electronic commerce, and information management.

In [53] an extension to the open-source model checker MCMAS [54] is presented (MCMAS-P) that is specialised for MA systems where there is an arbitrary number of agents in a system. Specifically, [53] introduces a technique for dealing with agents as parameterised interleaved systems. This technique identifies the maximum number of agents required in a model such that the results from its verification also apply to models with any larger number of agents.

Other formal approaches that involve the verification systems for MA programs based on the BDI framework include [55] and [56]. Their approaches involve getting the output of a BDI agent programming language into a format that can be verified by a model checker. In [56] the agent program model checker Agent Java PathFinder (AJPF) is used to generate Kripke structures which can then be verified by SPIN. The AJPF code is also used to generate DTMCs which are then able to be verified in PRISM. A comparison of verification results (runtimes) shows that converting a AJPF model and verifying properties with SPIN is more efficient than simply verifying the properties with AJPF.

Model checking with AgentSpeak

This section describes combining model checking with the AgentSpeak representation of an agent. Automatic construction of a model from an autonomous agent's specification in AgentSpeak is discussed in [57]; One of the initial hurdles

is that, unlike model checkers, AgentSpeak is not restricted to finite state systems. To overcome this, a new restricted version of AgentSpeak is introduced called AgentSpeak(F).

In [57], once an agent is represented in AgentSpeak(F) it can then be automatically translated into two modelling languages, PROMELA and Java PathFinder (JPF) [58]. From this, two separate models are generated and compared. The comparison shows a clear advantage of using PROMELA and SPIN opposed to JPF. This advantage is highlighted by the difference in time and memory to verify the same AgentSpeak(F) representation: verifying the system took SPIN 65.78 seconds using 210.51 MB of memory, and took JPF 18.49 hours using 366.68 MB of memory.

Alternate versions of AgentSpeak have been used for describing the behaviour of real-time agent-based systems, such as AgentSpeak(RT) [59]. AgentSpeak(RT) is further extended in [60] where it is also able to represent hard and soft deadlines within its BDI framework.

Extending BDI for learning The MA systems described here do not have a learning element: they are based on the BDI framework, which does not explicitly accommodate agents' learning. In the models generated, the implementation of the BRF function simply adds and removes beliefs based on the information from an agents' percepts, while the BRF's interpretation of perceptual information is unchanged.

It could be possible to extend this BDI framework to incorporate learning. One approach is to have a notion of variable beliefs, as opposed to beliefs based solely on current perceptual information. These beliefs would vary with new perceptual information and have an impact on the options available to each agent.

Some AgentSpeak interpreters have already been developed to incorporate predefined BRFs, which allows for more complicated belief revisions, e.g., resolving the perception of conflicting information in [61] is incorporated into the interpreter Jason [62].

UML representation

Having a high-level representation of an MA system provides more flexibility in what techniques can be used to analyse them. This is particularly true of model checking, where the systems must be abstracted to a relatively high level before checking properties. In this section we describe a method of representing an MA system in a UML framework.

In [63] an extension of UML that includes the representation of agents is presented. The aim is to create a unified framework for modelling MA systems. Extending UML in this way is not new, but here there are three key differences to previous approaches. These differences are (taken verbatim from [63]):

> "the proposal of a clear definition and representation of the elements that compose MA systems, the clear definition and representation of the relationships between the elements in static diagrams, and the representation of the interactions between all elements in a dynamic diagram."

Having this high level UML description as a standard way to describe MA systems could allow for automated translation into a modelling language. This proposal is based on having the extended UML as a well-defined input for a translator program. However, UML representation is designed for translation to OO programming languages, not modelling languages. This makes this extension better suited for creating software simulators rather than verifiable models.

Argumentation

Argumentation is a method for resolving a controversial standpoint. In an MA system this could relate to how an agent should proceed when only having limited information. Argumentation is a mechanism for forming and revising beliefs and decisions, as well as describing rational interactions in MA systems [64]. It is presented as an alternate method for making decisions based on incomplete information (nonmonatomic reasoning). It is more related to the decision process of an agent than to representing a whole system.

Grouping agents

Another direction of MA systems research involves treating a group of agents as a single entity. In [65] this is referred to as the *Gaia methodology*. The Gaia methodology is a standardised way of translating from a requirements capture to a grouped, agent model. For the Gaia methodology to be applied, there are three main assumptions placed on the system. These are that: agents always share common goals, there is a static organisational structure of the agents, and there is no uncertainty in the system.

The Gaia methodology promotes the use of fixed infrastructures, which gears it more toward industry, and particularly manufacturing; yet even regarding more fluid infrastructures, it can be used to provide a standardising process that moves from requirements to a model.

Similar work on treating a system of agents as a single entity is covered in [66]. Here a group of agents is treated as a new, different agent; this new agent is then modelled separately from the original agents. This approach is presented as a form of abstraction.

Pattern identification Further work on grouping agents involves identifying patterns within MA systems. In [67] a standardised way to identify *agent-oriented patterns* is introduced, where a pattern is identified for a problem specification of a system.

Defining a system of agents as a pattern allows them to be represented as one entity. This can allow for simplified analysis: if a pattern is known to satisfy certain properties then identifying this pattern in a system's specification allows one to infer that the same properties hold for this system. Models can also be simplified by being broken down into known patterns.

Some more work on the formal aspects of multi-agent systems is presented in an annual workshop *–Formal Aspects of multi-agent systems* (see, for example, [68]). Approaches tend to focus on protocol verification, formalisation of goals and plans, and knowledge-based agents.

2.6.3 Environment modelling

Modelling an environment is an intricate part of modelling any MA system because all interactions within an MA system must occur in its environment, and must abide by the laws imposed by this environment. In this section we describe techniques for representing an environment in an MA system.[6]

Classically, the environment was not seen as an explicit part of a system, but as an external part. This notion is contested in [70], where it is proposed that the environment is an explicit part of any MA system. An argument is presented advocating the inclusion of the environment in the system model. The general principle of the argument in [70] is summarised thus:

> "...the environment provides the surrounding conditions for agents to exist, which implies that the environment is an essential part of every multi-agent system."

Having an explicit representation of the environment means that an agent's actions can be described as *situated actions*; i.e., an agent's actions have different meanings depending on the state of the environment. A description of an environment in the case of an MA system, is given in [70] as:

> "The environment is a first-class abstraction that provides the surrounding conditions for agents to exist and that mediates both the interaction among agents and the access to resources."

Figure 2.38 illustrates this explicit representation (it is a simplified version of a figure from [70]). The dashed lines (both types) indicate data flow, and arrowed lines indicate interaction. The *deployment context* is the traditional view of the environment as an external set of resources that an MA system interacts with. The *application environment* is a layer between the agents and the deployment context, which acts as an interface. The application environment is split into distinct

[6] An accumulation of research on the topic of environments in MA systems was published in the Autonomous Agent and Multi-Agent Systems journal 2007, and summarised in [69].

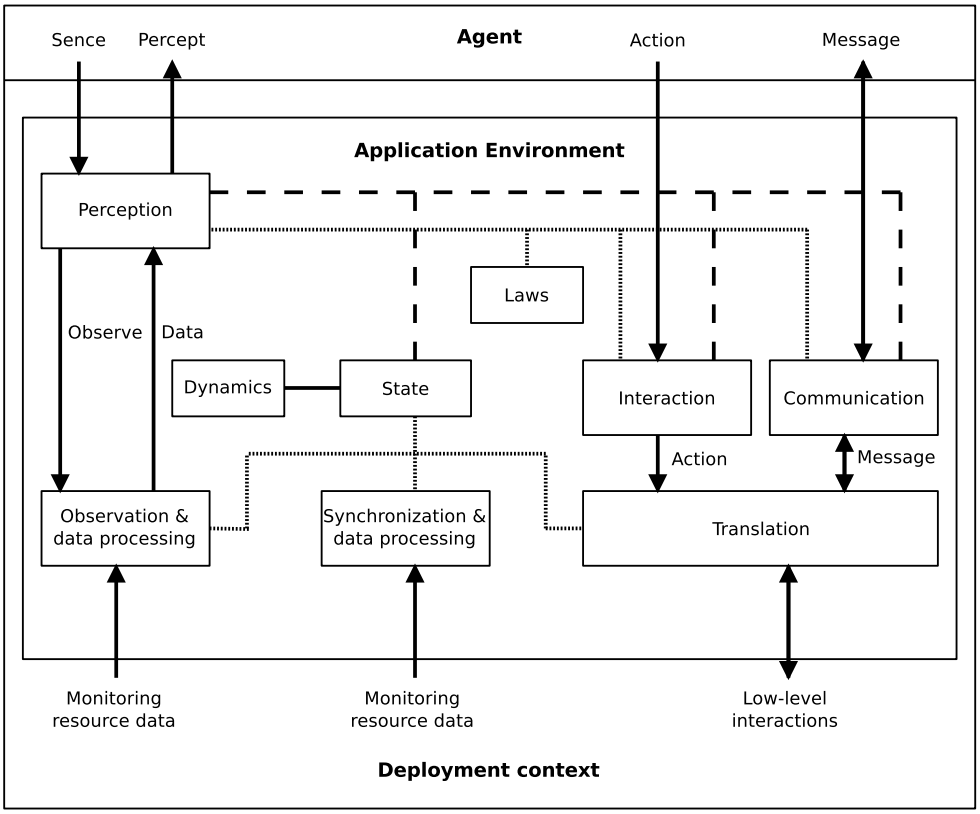


Figure 2.38: Explicit representation of an MA system's environment.

components, which have their values combined to give the current state of the system.

The data flow of information shows how an agent's perception of the environment is affected by the following: its communication with the environment, its interaction with the environment, the *state* of the environment, and the environment's *laws*. This representation takes into account dynamic properties (*dynamics*), which directly affect the state of the environment; it also takes into account the effect of monitoring the deployment context, which indirectly affects the state of the environment.

An alternate, explicit representation of an environment is given in [71], where an environment is formally defined with set notation. This definition is composed of four components. These are: structural, dynamism, agents manipulating dynamism, and interference with dynamism. The *structural* component represents

the state of the environment, and it is decomposed into the following four parts: *entities* (these are objects within the system e.g., an agent), *properties* (these are measurable system values e.g., temperature), *embodiments* (how an agent relates to the environment e.g., an agent embodies a robot), and *constituents* (these are sets of entities unioned with sets of properties).

The *dynamism* of the environment is used to express the evolution of environmental entities. It is decomposed into activities which involve a set of constituent entities, a time interval, and an evolutionary strategy. An evolutionary strategy represents an activity's effect on the set of constituent entities over time.

The *manipulation of dynamism* describes how agents are able affect the activities of the environment according to *reaction laws* (a set of logical rules governing environmental interactions). The *interference with dynamism* describes how the different elements of dynamism interact with one another, and specifies the ways in which entities interfere.

The body of work on environmental representation suggests that formally defining the environment is necessary to accurately model an MA system; yet, there is still no overriding standard for representing an environment. Therefore, decomposing an environment into its components is a different process for each system. The decomposition of an environment presented in [70] and [71] is used in order to generate a software simulation, rather than a model that can be verified. These approaches are too detailed to be useful in a model checking context.

2.6.4 Representing learning in MA systems

In this section we give a summary of approaches to dealing with learning agents in MA systems.

A comparison between two commonly used learning techniques, *temporal difference* (TD) and *evolutionary* learning methods is presented in [72]. More specifically, the Sarsa (TD) and NEAT (evolutionary) methods are compared; where both methods have been shown to be empirically successful [72]. Our interest lies in the idea of trying to create a benchmark test for learning techniques. The comparison of the techniques is trial-based, where the context of the learning is

of significant importance (benchmark test). Benchmark testing for learning algorithms, in this way, helps provide an appropriate context for modelling them. Defining a good benchmark test is essential for fair comparisons between different methods. The type of ABL system we model can act as a benchmark for implementing other algorithms in place of the ICO learning that we currently consider.

MA cooperative agent learning

In this section we discuss the issues that relate to coordinated actions in MA systems.[7] Here we refer to the issues that arise when coordinated actions occur in MA systems as the *coordination problem*.

The coordination problem arises when different agents have to coordinate to achieve a common goal. In [74] reinforcement learning is used to deal this problem. The agents' behaviours are affected by a *course-grained* and a *fine-grained* algorithm in order to converge to a common behaviour that will enable them to achieve their goals. These algorithms are formally shown to achieve convergence, and can be used to predict the steady-state behaviour of system.

Further work on the coordination problem is described in [75]. It describes distributed learning techniques that improve coordination among autonomous agents.

Cooperative learning is applied to modelling infinite state-space MDPs in [76], with an aim to cause agents' behaviours to converge into a predictable optimal policy behaviour. Here a formula is used to calculate whether the agents' behaviour will converge to this.

Identifying convergent behaviour in MA systems allows (model checking) verification to be applied only to the convergent behaviour –as opposed to the pre-convergent behaviour. For example, suppose there are two robots each with a set behaviour. By calculating the convergent behaviour before generating the model, we can create a model that simply represents the convergent behaviour and only apply model checking to that. This approach can reduce the size of a model provided that properties of the pre-convergent behaviour do not need to be verified.

[7] A broad survey of literature on cooperative agent learning in MA systems is presented in [73].

Chapter 3

Preliminary ABL models

In this chapter we describe the modelling and verification of three ABL systems in PROMELA and SPIN respectively. Each physical system is explained using the term robot, and its model is explained using the term agent. This is a necessary distinction between the real system, which uses actual robots, and the models, where these robots are represented by software.

We then present our PRISM models. The first two of these are based on the same systems as those described in the PROMELA models. The others focus on modelling learning specifically, where the first two are based on theoretical systems, while the final system involves a learning robot, designed for obstacle avoidance.

All the models in this chapter are preliminary models in our research. Most are simpler versions of the final ABL system models covered in Chapter 4 and Chapter 6.

3.1 PROMELA models

In this section we describe three systems and models that represent incremental steps in the generation of the Explicit model (see Chapter 4) of the ABL system we described in Section 2.2.3. First, we describe a basic system model, which involves two agents trying to navigate in a small, enclosed environment without colliding into one another. In Section 3.1.2 we present a slightly more complicated

version of the scenario that we call *colliding robots*, in which the agents have a *avoidance field* that they use to send information to one another. We then describe a model of a system that involves robots that use dual antennas to navigate an environment and avoid collisions. Finally, we sum up the merits and drawbacks of the preliminary models and explain how we plan to proceed.

3.1.1 Colliding robots

In this section we consider a system that involves two robots trying to avoid one another in a walled environment. Each robot has a proximal sensor that tests for the presence of an obstacle directly in front of it; these sensors are used to avoid colliding with obstacles, other robots, or the perimeter wall.

We begin by modelling each agent and the environment as processes. The environment is represented as a grid where coordinates are stored in a global, two-dimensional array; where each element of the array is indexed by a coordinate of the grid and describes what is held at that coordinate. For this model we use a 8×7 array. The grid is surrounded by a perimeter wall, which is stored in the array as a wall of obstacles. The environment process has access to the location of the agents and the array of grid cells. An agent communicates with the environment via channels, and the environment responds using a different channel for each agent. (The PROMELA data structures we refer to here, processes and channels, are described in Section 2.4.1.)

Agents move by choosing nondeterministically whether to move up, down, left, or right. When an agent chooses an adjacent grid cell to move to, it first turns to face this direction and uses its proximal sensor to test if the grid cell is unoccupied. This test is done by sending the environment a message that contains the coordinate that the agent is trying to move to, via a channel; where this channel is used to represent a robot's proximal sensor probing the environment. The environment responds with either "`clear`" or "`nogo`". This indicates whether the adjacent cell contains an obstacle or not. If the adjacent cell is occupied, the robot again chooses an adjacent cell (turns) and checks it. When an empty cell has been located, the agent moves into that cell. The agents proceed to move in this fashion

–randomly–, with the only goal of trying to avoid collisions.

The MSC in Figure 3.1 shows how the agents and environment pass messages. The full code of this model is given in Appendix A.1.

Assumptions

We make assumptions about this system to simplify our PROMELA models, they are: that an agent can sense only one grid cell in front of it (in the direction it is facing), and that an agent's proximal sensor detects obstacles, other agents, and the perimeter wall.

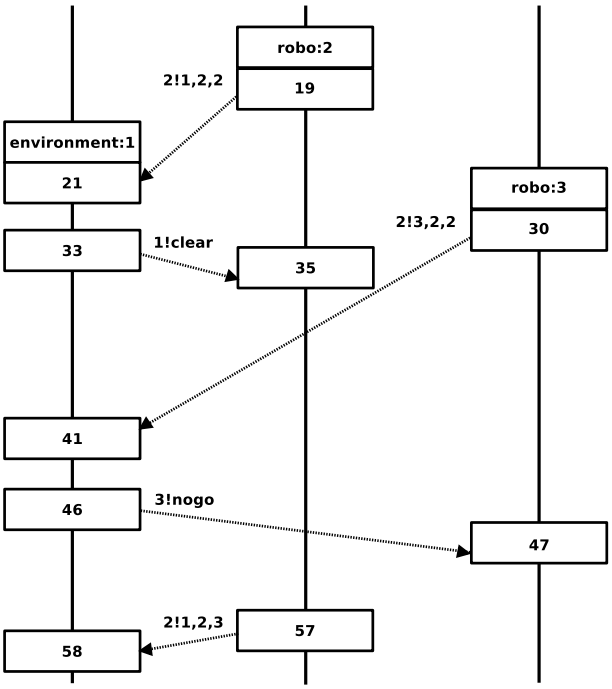


Figure 3.1: MSC for Colliding robots.

Figure 3.1 represents the communication between two agents interacting with their environment, namely `robo:2`, `robo:3`, and `environment:1` respectively (MSCs are described in Section 2.4.1). Each agent sends a message to the environment using its proximal sensor, and the environment responds. Specifically, `robo:2` sends a message querying whether the cell with coordinate `(2,2)` is free; the response `clear` is sent back, and then `robo:2` moves to that coor-

dinate. The agent `robo:3` then queries whether `(2,2)` is free and receives a `nogo` message, because the coordinate is now occupied.

Verification

In the real system it is impossible for these two robots to collide because they are each checking for potential collisions continuously and independently. To make our model a closer representation of this system, it is necessary to check to see if we can replicate this perfect avoidance behaviour in our model.

The following proposition definitions (in 3.1) and LTL formula (3.2) specify the property of perfect avoidance behaviour for this model.

```
define  p  roboX[0] == roboX[1];
define  q  roboY[0] == roboY[1];          (3.1)
define  v  validator == 1;
```

```
    []  ( !  (p  &&  q  &&  v))           (3.2)
```

Formula 3.2 expresses the property that it is always false that the agents share the same coordinates –once the systems has been initiated (validated). It tests whether the agents' coordinates are the same at any time step. Variables `p` and `q` are true if the agent's coordinates are equal, while `v` is true once the system is instantiated. The validator makes sure that the test is not carried out until the `roboX` and `roboY` arrays have been assigned values, where these arrays hold the coordinates of the agents. These arrays are indexed by an agent's identification number, such that `roboX[0]` is the `x` coordinate of agent number `0`. The full verification output is in Appendix A.2.

Time-step jumping problem The property expressed by Formula 3.2 (perfect avoidance behaviour) does not hold for all paths of our model; hence, the verification shows the property to be false (we refer to this as *failed verification*). Examination of a counterexample, produced by SPIN, shows how the verification fails. We refer to the cause of this failed verification as the *time-step jumping*

problem (similar to the instantaneous transition problem in Section 2.4.3).

This problem occurs when both agents approach each other as shown in Figure 3.2, and then both select the same coordinate to move to. Each agent is able to receive a `clear` message from the environment before they move, which means that they are able to collide –occupy the same grid cell in the model.

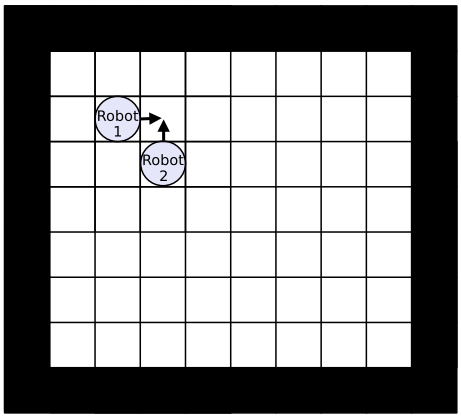


Figure 3.2: Example of the time-step jumping problem.

This problem highlights the issue of abstracting the environment to a grid representation. Each movement in a grid is a jump, while in reality it is a gradual process. In the actual system the robots' sensors are continuously testing for obstacles as the robots begin to move into a cell; where, due to this continuous testing, the robots recognise the potential for a collision as they are moving into the new grid cell and turn to avoid, before colliding.

This issue can also be caused by the shape of the agents and their sensor range. For example, if the agents are circular, as shown in Figure 3.2, then the agents can notice one another before the full movement into the gird cell takes place. Yet, if the agents are square then their corners will collide as soon as they move, even the smallest amount, into the new grid cell. This is simply an alternate cause of the same problem. The real issue remains: we need to represent the real system in the model, and the robots can always avoid each other in the real system.

Another issue highlighted by this failed verification is with the use of buffered channels. Between a message being sent and received it is possible for another process to execute a transition. For example, suppose a robot sending a message

enquiring if a cell is free. While the `clear` message is being sent back, it is possible for the other agent to move into the cell between the sending and receiving of the message.

A solution to this problem that helps more accurately model the ABL system is to have an additional state per grid cell that represents the situation when an agent is approaching that cell. When the agent is in this *approaching cell* state, it performs another test to see if the cell is clear. This test includes checking that no other agents are in the approaching cell state for that cell. In this way, two agents cannot be in the same cell. This solution emulates the *synchronisation points* described in Section 2.4.3, in that both agents have to synchronise on a transition into a new cell. This solution is modelled and checked with the same definitions and formula as before. The property is now true.

Verifying this property demonstrates that the agents cannot now occupy the same cell at the same time. Hence, the real robot's behaviour is more accurately represented, but at the cost of additional complexity in the model.

The verification output provided by SPIN is shown in Figure 3.3. The first three lines indicate the version of SPIN used, and the fact that POR and compression are applied (see Sections 2.3.13 and 2.3.13 respectively). The term "`never claim`" is described in Section 2.4.1. The "`states stored`", "`depth`", and the "`total actual memory usage`" are also significant. The states stored corresponds to the number of states encountered for the DFS of the state-space, and the depth reached refers to the length of the longest path explored. If the models or LTL formula get more complicated the amount of memory usage can become a limiting factor when trying to run verifications. The full code for the agents with the approaching-cell state is given in Appendix A.3.

3.1.2 Avoidance field robots

In this section we describe a system involving two robots in a walled environment. Each robot has a sensor that allows the robot to test an adjacent square for the presence of an obstacle or an *avoidance field*. An avoidance field is a area surrounding each robot. The field is used by the robot as an avoidance signal, and

```
(Spin Version 5.2.2  7 September 2009)
        + Partial Order Reduction
        + Compression

Full statespace search for:
        never claim             +
        assertion violations    + (if within scope of claim)
        acceptance   cycles     + (fairness disabled)
        invalid end states        (disabled by never claim)

State vector 63 byte, depth reached 416528, errors: 0
  2247944 states, stored
  1476357 states, matched
  3724301 transitions (= stored+matched)
        1 atomic steps
hash conflicts:   2988721 (resolved)

Stats on memory usage (in Megabytes):
  177.936   equivalent memory usage for states (stored*(State vector +
overhead))
   81.659   actual memory usage for states (compression: 45.89%)
            state vector as stored = 18 byte + 20 byte overhead
    8.000   memory used for hash table ( w21)
  305.176   memory used for DFS stack ( m10000000)
  394.668   total actual memory usage
```

Figure 3.3: Verification output: agents not colliding by using approaching-cell state.

it is employed in order to help prevent collisions. The environment is represented as a 8×7 grid, surrounded by a perimeter wall. The robots aim to avoid colliding with obstacles, the perimeter wall, and other avoidance fields.

Assumptions

We assume that: an agent can sense only one cell in front of it (in the direction it is facing), and an agent's sensor detects obstacles, other avoidance fields, other agents, and the perimeter wall. Avoidance fields surround and cover each agent as

a 3×3 grid, with the agent at its centre.

System model

The MSC in Figure 3.4 shows that the agents and the environment interact similarly to the previous model. The agents "`robo:2`" and "`robo:3`" send messages to the environment "`environment:1`", which responds with a "`nogo`" or "`clear`" message. Each message that an agent sends consists of its response channel, the coordinates it is moving to, and its id.

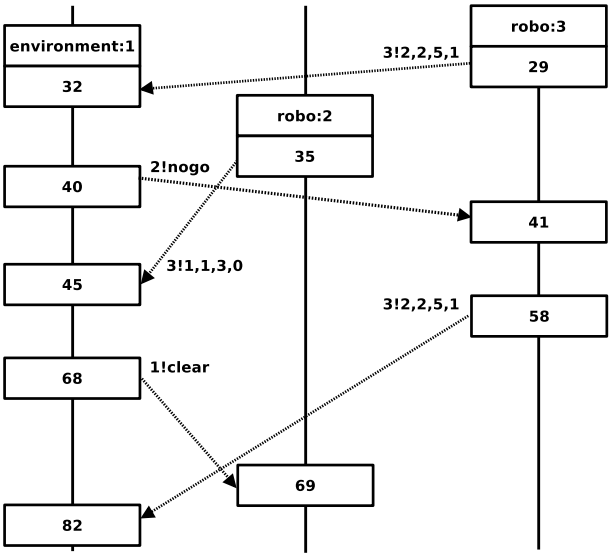


Figure 3.4: MSC of agents with avoidance fields.

Figure 3.5 represents the two agents and their avoidance fields in the environment. Note that the avoidance field of the second robot overlaps with the perimeter wall.

Verification

We want to verify that the addition of avoidance fields, and sensors to detect them make it impossible for the agents to collide. The property used is the same as that used in the previous model.

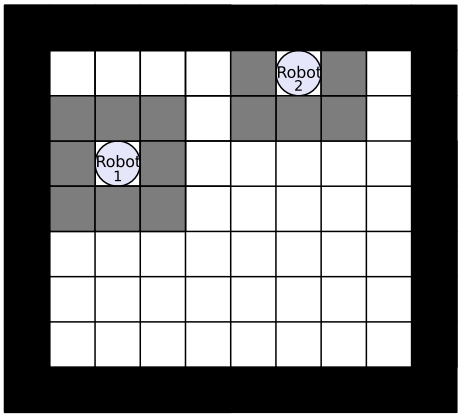


Figure 3.5: Agents with avoidance fields.

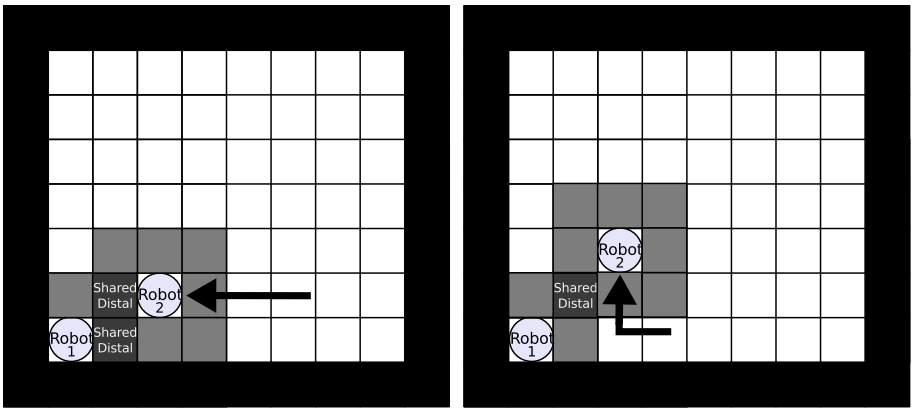


Figure 3.6: Example of the agents' avoiding colliding with their avoidance fields.

The time-step jumping problem discussed in Section 3.1.1 is no longer an issue with this model. This is because the agents are now unable to move to a position diagonally beside each other, as shown in Figure 3.6, without detecting and then avoiding the other's field. The full code for the Avoidance fields model is given in Appendix A.5.

3.1.3 Dual antenna robots

The third system we model involves two robots that are trying to avoid one another using a dual antenna system. The proportions of the agents are set to a more accurate scale than in the previous models. In order to get the scale more accurate

the resolution of the environment's grid is increased to a 22×22 array, where the outer grid cells form the perimeter wall. Each agent covers an area of 2×2 cells, with distal antennas four cells long and proximal antennas one cell long. The antennas are arranged as pairs, a left and a right, each containing one proximal and one distal antenna. As in previous models the agents try to navigate their environment without colliding into anything.

System model

The MSC in Figure 3.7 demonstrates how the agents interact with each other and the environment. The agents are represented by processes `robo:3` and `robo:4`. Each agent sends messages to the environment via its proximal and distal antennas. The process `sensorLong:2` represents the agents' antennas (all long and short range antennas). When an agent receives a message of "`clear`" back from its antennas (`sensorLong:2`) it processes the message, and then tries to move forward to a new cell. In order to move, a message is sent to the `environment:1` process. Note that this process is different from the previous models because here it is used to synchronise the agents' movements. This means that an agent cannot move into a new grid cell without the cell being empty.

In this model the agents are much more closely based on the definition of an agent from Section 2.2.1; where sensors and actuators are treated separately from the agent. Actuators act as a bridge between an agent process (the agent's internal processing) and the environment's process –relaying to the environment what action that agent is taking. The sensor processes each hold a perception of the environment which they relay to the agent processes. Agents' decisions are made based on information from the sensors.

The number of directions that an agent can be orientated is increased to eight, from four in previous models. This increase allows all adjacent grid cells to be accessed by an agent. Even though it is possible to measure the direction an agent is facing to a single degree, a further increase in accuracy would greatly increase the state-space of the model. It would also require a higher resolution of grid to allow for the new turning and movement precision. Alternatively, the agent's

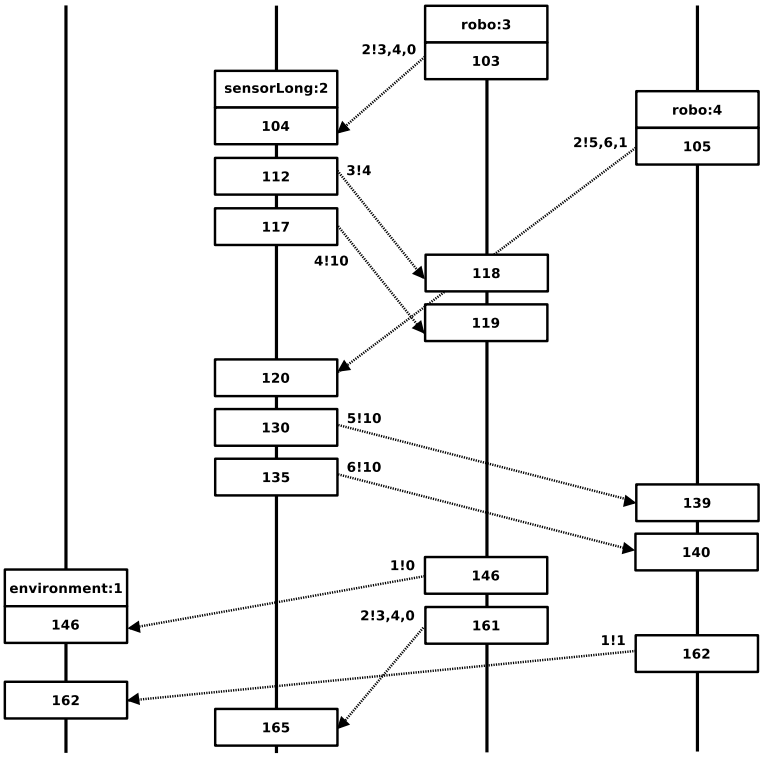


Figure 3.7: MSC of dual antenna robots.

movement could be estimated to be one of the eight adjacent grid cells based on the angle that the agent is facing. (This is essentially the same thing as still using only eight directions.)

Instead of static, cell-to-cell movements like in the previous models, this model is designed to better emulate the continuous driving of real robots –they make small adjustments as they move. Each agent reassesses its direction every time it arrives at a new grid cell, and having a higher resolution of the grid means that calculations for a new move are performed at much smaller intervals than before. Figure 3.8 illustrates the agents in their environment.

The calculations for an agent's movement are based on the difference in the signals from the antennas. The antennas' function is as described in Section 2.2.3, where each antenna produces varying strength signals depending on how close an obstacle is to it. Once the agent processes the antennas' signal, it turns to avoid the closest obstacle. This emulates the avoidance behaviour of the real robots. An

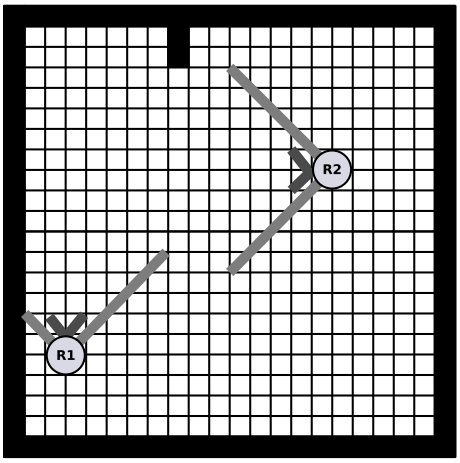


Figure 3.8: Example of agents with dual antennas.

abridged version of the PROMELA code for this model is given in Appendix A.6.

Assumptions

Antennas are treated as perfect springs, which allows us to neglect their resistance on the agent when they contact an obstacle. Therefore, whenever an antenna is contacted by an obstacle it contracts in length to the distance from its base to the obstacle. This happens without impacting on the position of the obstacle or the agent. It is important to note that, because of this contraction, only one obstacle can be sensed on a pair of antennas (left or right) at a time. For an agent, an obstacle is anything in the environment other than free-space. If there is more than one obstacle along the line of an antenna, then the closest obstacle provides the signal to the agent (as the antenna has contracted to that distance).

Representing the environment as a gird while having long antennas on the agents introduces discrepancies in the size of the antennas in different situations. For simplicity, we assume that these discrepancies are negligible.

These discrepancies are shown in Figure 3.9. They are apparent when an agent is facing in different directions, where they occur in both the angle at which the agent's antennas project and in their length and size. The angle of projection changes slightly between the directions that an agent can face. This change can

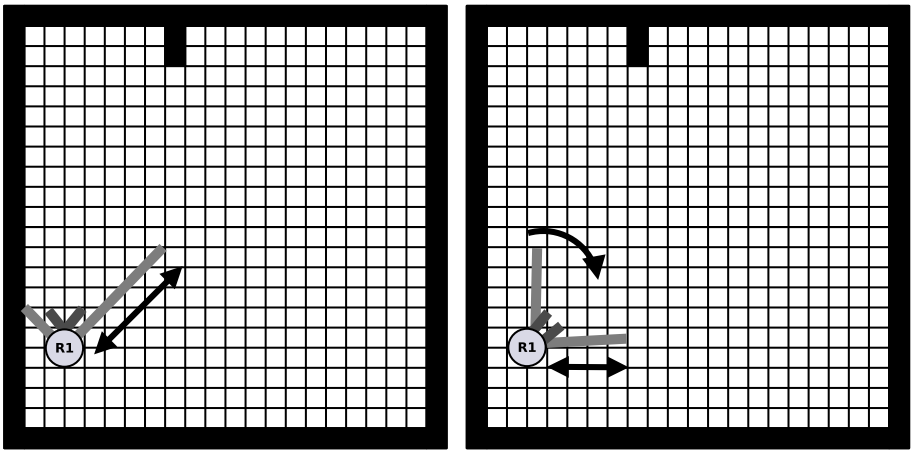


Figure 3.9: Agent turning 45° clockwise

be seen by comparing the agent facing North to it facing North-East. Additionally, when the agent faces diagonally, its proximal antennas cover three grid cells, as opposed to two. The increase in antenna coverage allows the detection of objects directly in front of the agents, as with the real system. The discrepancy in antenna length is shown comparing the distal antennas of the two agents. It is due to the diagonal distance across a grid cell being greater than the distance across it horizontally or vertically.

The process of an agent's learning is not considered in this model. Here, we are concerned with modelling the system when the agents have a fixed level of knowledge. In the real system, the agents begin by not knowing how to interpret signals from their distal antennas, then learn how to interpret them to avoid colliding with obstacles. All the following verifications are done with the assumption that the agents have learnt how to respond to their distal antennas.

Verification

As this model is more complex, its verification is a much more memory intensive process than for the previous models. The greater number of variables create a much larger state-space, which requires us to refine the PROMELA code to be as efficiently expressed as possible before we verify the model. This involves identifying all situations where the state-space can be reduced by combining states

using e.g., *atomic statements* (see Section 2.4.1). Note that for the following verifications, when we refer to colliding with obstacles this includes the perimeter wall, but not the other agent.

The first property we verify specifies that: an agent never crashes into obstacles, which is represented by Formula 3.4 (referring to the Definitions in 3.3).

```
define  a  roboX[0];
define  b  (roboX[0] + 1);
define  c  roboY[0];
define  d  (roboY[0] + 1);
define  e  (xAxis[a].yAxis[c] == 1);
define  f  (xAxis[a].yAxis[d] == 1);
define  g  (xAxis[b].yAxis[c] == 1);
define  h  (xAxis[b].yAxis[d] == 1);
```
(3.3)

```
[]  !  (e || f || g || h)
```
(3.4)

Variables `a`, `b`, `c`, and `d` are used to store the agent's coordinates. Variables `e`, `f`, `g`, and `h` are propositions to determine whether the agent has collided with any obstacle. The term `xAxis[a].yAxis[c]` denotes the array element that is a grid cell `a` along the x-axis and `c` up the y-axis. The propositions evaluate to true if any portion of the agent occupies a cell for which the associated array element is set to 1.

The second property is: the agents do not collide with each other or any obstacle, which is represented by Formula 3.7 (referring to the Definitions in 3.5 and 3.6).

```
define a roboX[0];        define i (xAxis[a].yAxis[c] == 1);
define b (roboX[0] + 1); define j (xAxis[a].yAxis[d] == 1);
define c roboY[0];        define k (xAxis[b].yAxis[c] == 1);
define d (roboY[0] + 1); define l (xAxis[b].yAxis[d] == 1);
define e roboX[1];        define m (xAxis[e].yAxis[g] == 1);
define f (roboX[1] + 1); define n (xAxis[e].yAxis[h] == 1);
define g roboY[1];        define o (xAxis[f].yAxis[g] == 1);
define h (roboY[1] + 1); define p (xAxis[f].yAxis[h] == 1);
```
(3.5)

```
define q ( (xAxis[a].yAxis[c] == xAxis[e].yAxis[g]) ||
           (xAxis[a].yAxis[c] == xAxis[e].yAxis[h]) ||
           (xAxis[a].yAxis[c] == xAxis[f].yAxis[g]) ||
           (xAxis[a].yAxis[c] == xAxis[f].yAxis[h]) )
define r ( (xAxis[a].yAxis[d] == xAxis[e].yAxis[g]) ||
           (xAxis[a].yAxis[d] == xAxis[e].yAxis[h]) ||
           (xAxis[a].yAxis[d] == xAxis[f].yAxis[g]) ||
           (xAxis[a].yAxis[d] == xAxis[f].yAxis[h]) )
define s ( (xAxis[b].yAxis[c] == xAxis[e].yAxis[g]) ||
           (xAxis[b].yAxis[c] == xAxis[e].yAxis[h]) ||
           (xAxis[b].yAxis[c] == xAxis[f].yAxis[g]) ||
           (xAxis[b].yAxis[c] == xAxis[f].yAxis[h]) )
define t ( (xAxis[b].yAxis[d] == xAxis[e].yAxis[g]) ||
           (xAxis[b].yAxis[d] == xAxis[e].yAxis[h]) ||
           (xAxis[b].yAxis[d] == xAxis[f].yAxis[g]) ||
           (xAxis[b].yAxis[d] == xAxis[f].yAxis[h]) )
```
(3.6)

```
[] !(i || j || k || l || m || n || o || p || q || r || s || t)
```
(3.7)

This property is also shown to be true. Success of the final verification shows that once the agents have learnt to use their distal antennas they are able to avoid all collisions in this type of environment.

3.2 PRISM models

In this section we present a variety of PRISM models.

3.2.1 Colliding robots

This model involves two agents that can detect an obstacle in the square in front of them. There are four variations of the agents considered here. Two variations where the agents move asynchronously; the first where they move in four directions, and the second in eight directions. The other variations are where the agents are forced to move synchronously; the third in four directions, and the fourth in eight directions. Checking different variations allows us to assess how each one affects the properties of the system.

Here, the agents are in a square grid, surrounded by a perimeter wall composed of obstacles. We use a different size of grid from the Colliding robots in Section 3.1.1. For verification we use a 22×22 grid. This matches the same scale and size as the Dual antenna robots in Section 3.1.3.

Assumptions

Each agent can only move forward one grid cell in one time step; and when an agent detects an obstacle, it can turn either left or right. Unlike the SPIN models, turning left or right is no longer a nondeterministic choice, as the agents now have 0.5 probability for turning left and 0.5 of turning right.

Apart from the addition of probabilistic choice and the two 8-directional variations, these models adhere to the same assumptions as with the Colliding robots SPIN model (for details see *Assumptions* in Section 3.1.1).

Verification

The initial verification is carried out on two variations. These are where agents move asynchronously, with one using four directions of movement, and the other eight directions. We check the $PCTL$ Formula 3.8, which queries the probability of the agents eventually occupying the same grid cell.

```
P = ?  [ F  [ (x1 = x2)  &  (y1 = y2) ] ]
```
(3.8)

For both variations, the probability of them eventually occupying the same cell (colliding) is 0.0. This emulates the perfect avoidance behaviour of the latter SPIN Colliding robots model.

Although these verifications are successful, a more interesting verification is to assess whether this property holds when the agents are forced to synchronise their movement; i.e., using the second two variations. Checking these variations with the same $PCTL$ formula gives the probability of the agents eventually colliding as 1.0.

After deriving that the agents in the second two variations are able to collide, we can now query the probability of them colliding, at any given time step, when the model is in a steady state (see Section 2.4.2). This is queried by the $PCTL$ Formula 3.9.

```
S = ?  [ (x1 = x2)  &  (y1 = y2) ]
```
(3.9)

Forcing the agents to synchronise their movements gives the steady state probabilities of both agents occupying the same grid cell (having collided) as: 0.0104 for the 4-directional agents, and 0.0064 for the 8-directional (to four decimal places).

Analysis

Querying Formula 3.9 over a range of grid sizes generates the graphs in Figure 3.10 and Figure 3.11. They show the probability of agents colliding as the size of the grid ($n \times n$) ranges from where $n = 2$ to $n = 20$.

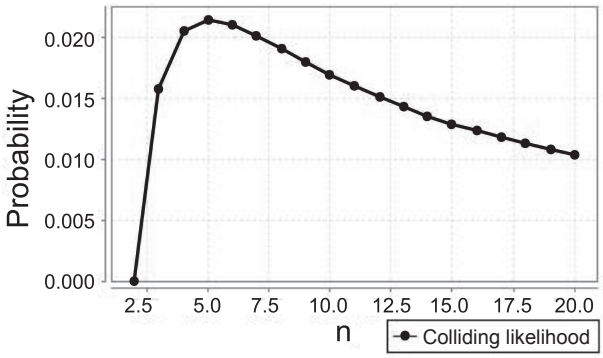


Figure 3.10: 4-directional agents, probability of colliding.

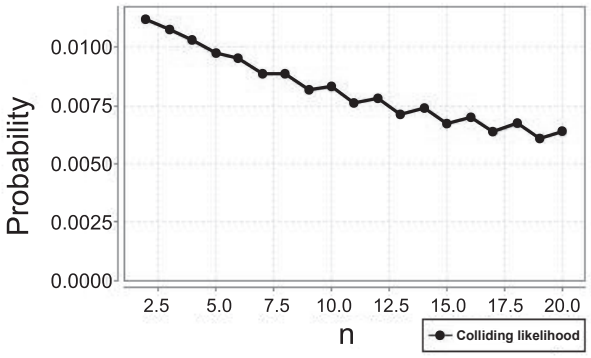


Figure 3.11: 8-directional agents, probability of colliding.

As the grid size increases, the likelihood of a collision decreases. The non-linear relationship between size and likelihood is due to the way the agents navigate. By constantly trying to move forward, they tend to move to the perimeter wall of the environment and then simply navigate around it. Note that, in Figure 3.10 there are no collisions in the smallest grid simply because the agents are unable to move. An abridged version of the PRISM code for this model is in Appendix B.1.

3.2.2 Dual antenna robots

This model is the PRISM equivalent of the Dual antenna robots SPIN model, from Section 3.1.3. As with the PRISM Colliding robots, we model the same four variations of robot. However, we are only interested in the variations where the agents are forced to move synchronously. The other variations are excluded because, as previous verifications show, their agents do not collide –even with only a basic sensor.

The assumptions here are the same as with the SPIN model, but with the same discrepancies as the PRISM Colliding robots model has with its equivalent SPIN model. Here, we query the probability of the agents colliding using the following $PCTL$ formula.

```
P = ?  [ ((x1=x2) & ((y1=y2) | (y1=(y2+1)))) | ((x1=(x2+1)) &
       ((y1=y2) | (y1=(y2+1)))) ]
```

(3.10)

Checking Formula 3.10 shows that the probability of the agents colliding is 0.0. As with the equivalent SPIN model, when the agents have the long distal antennas they cannot collide. An abridged version of the PRISM code for this model is in Appendix B.2.

3.2.3 Learning models

The first system in this section is loosely based on an example from [5]. Here we create rudimentary learning agents using PRISM. The agents differ from our definition in Section 2.2.1. Here, actuators and sensors are not explicitly represented. However, they are implicitly represented; e.g., the agent picks a bean from a bag (actuator), and identifies the colour (sensor).

The final model applies the same approach to model learning, but applies it to our ABL system, described in Section 2.2.3. It involves an agent learning to avoid obstacles in an open environment.

Bean bag prediction

For these models, agents are trying to assess the colour ratio of beans in a bag. There are two possible colours, blue or red, and there is a predefined ratio of the colours in each bag. The agents try to use the knowledge of what colours they have already seen, in order to predict the ratio of colours for that bag.

Bag ratios are: 100% blue, 70% blue and 30% red, 30% blue and 70% red, and 100% red. After a certain number of beans are picked, the agent predicts the ratio. Their selection is limited to the four possible ratios, stated above.

Here, the agent's prediction involves counting the number of red beans picked, and remembering the total number of beans picked. In Table 3.1, `redCount` refers to the number of red beans counted, and `beanTotal` the total. Once a given number of beans has been picked, the agent applies the following calculations.

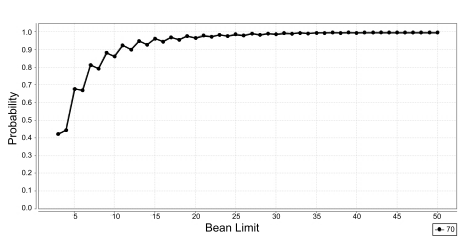
If (calculation)	Then (prediction)
`redCount = 0`	`100% blue bag`
`0 < redCount < (beanTotal/2)`	`70% blue bag`
`(beanTotal/2) <= redCount < beanTotal`	`70% red bag`
`redCount = beanTotal`	`100% red bag`

Table 3.1: Bag prediction calculations.

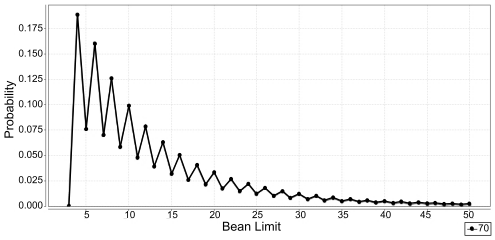
This prediction emulates a robot using the BDI representation of learning. The current beliefs of the agent are represented by which bag ratio it believes it is taking beans from. The desire of the agent is to identify the correct type of bag ratio, and its intention is to pick beans to decided the ratio. It revises its beliefs after each new bean is picked.

We assess the ability to predict the correct bag ratio after every bean picked. Figure 3.12 shows four graphs where the probability of the agent predicting each type of bag ratio is plotted against the number of beans picked from the bag. These graphs represent the bag ratio of 70% blue and 30% red.

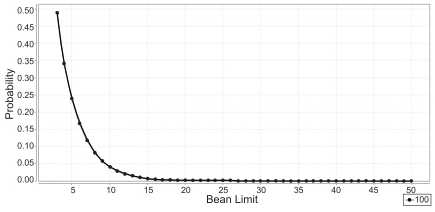
As more beans are picked, the probability of the agent predicting the correct ratio increases; i.e., probability of choosing a 70% blue bag, the correct answer,



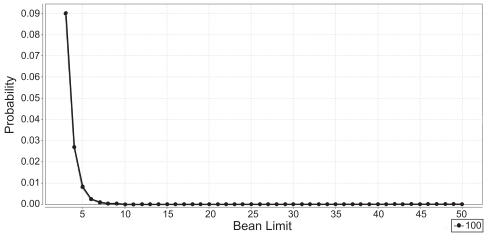
(a) Predicting 70% blue (correct choice).



(b) Predicting 70% red.



(c) Predicting 100% blue.



(d) Predicting 100% red.

Figure 3.12: Probability of correctly predicting bag: 70% blue, 30% red beans.

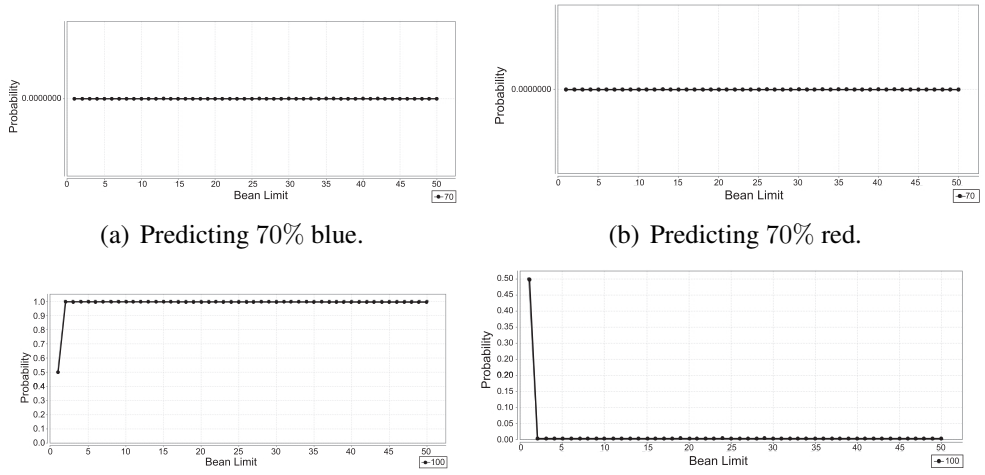
tends to 1.0 as the *Bean Limit* increases.

In Figure 3.13, the graphs represent the bag ratio of 100% blue. The same trend as with the previous bag ratio is observed here.

Analysis These models highlight the type of analysis that can be performed on a rudimentary learning system. Although the system is relatively simple, it still represents an agent with memory, and demonstrates how the prediction method can be quantified over different bag sizes and ratios. It is this quantification of properties that is of interest, as it allows direct comparison of prediction methods (learning methods). The PRISM code for this model is shown in Appendix B.3.

Learning obstacle avoidance

For this model we represent a robot that is learning to avoid obstacles in an open environment. Instead of explicitly representing the environment, an agent simply has a probability of encountering an obstacle when it moves forward. We are not as interested in proving that the agent will always learn to avoid obstacles,



(a) Predicting 70% blue.

(b) Predicting 70% red.

(c) Predicting 100% blue (correct choice).

(d) Predicting 100% red.

Figure 3.13: Probability of correctly predicting bag: 100% blue beans.

but in quantifying aspects of the system. For example, what is the probability of the agent doing something –as opposed to whether it will eventually or never do something. Specifically, we are calculating the most efficient angle to turn for a type of environment.

Learning is represented by the assessment of the efficiency of a chosen turning angle. Each choice is assessed probabilistically over a given number of obstacle encounters. The agent assesses which angle has been most efficient over the set number of encounters –in some situations several angles my be equally efficient. We assess two variants of this model; one where efficiency is determined by the energy consumption of the agent, and the other where it is determined by the avoidance of collisions.

In the first variant of the model we assess efficiency by the amount of energy the agent expends to avoid an obstacle. The more the agent has to turn to do this, the more energy it uses. When an agent collides with an obstacle it has to turn a full $90°$ to continue moving forward. The degree of the angle turned is used as the amount of energy expended by turning. Hence, if the agent can avoid an obstacle by turning $30°$ before a potential collision then it will save 60 units of energy (as opposed to $90°$ from a collision).

We measure the efficiency of the agent's choice in different environment types. Each type has a different probability of encountering an obstacle, where there are two different sizes of obstacle. These are: a large obstacle which requires a large turn, and a small obstacle which requires a small turn. There is also a different probability associated with encountering each type of obstacle. The efficiency we are interested in is for specific environment types; i.e., environments with different ratios of large to small obstacles.

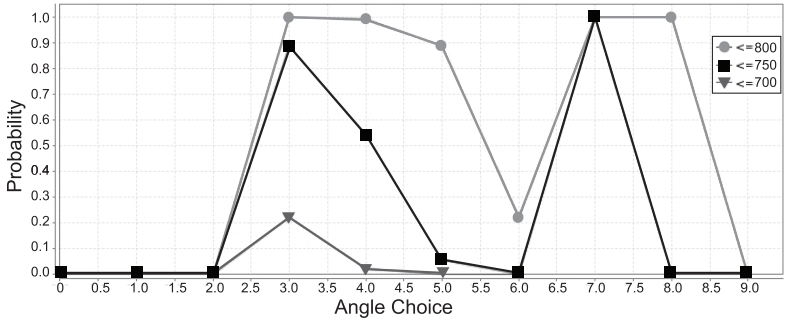
Learning the optimal turning angle is not just an exercise in choosing the smallest angle which avoids all obstacles, as the most efficient turning angle for a type of environment need not avoid all obstacles. In fact, the most efficient choice of angle is based on the frequency of different types of obstacle. For example, suppose an agent is in an environment where there is a majority of easily avoided obstacles with few requiring large angles to avoid. Here it may be more efficient to use a small turning angle for this type of environment, and just collide using the full 90 units for the few large obstacles.

The graphs in Figure 3.14 show the most efficient turning angles to choose. The efficiency is based on the amount of energy expended after 100 obstacles have been encountered by the agent. Each plot in the graphs represents an energy level. The plot representing $<= 800$ represents the probability of the agent expending, at most, 800 units of energy for the different turning angles (x-axis). On the x-axis a value x represents an $x \times 10^{\circ}$ turn. Each graph represents a different type of environment.

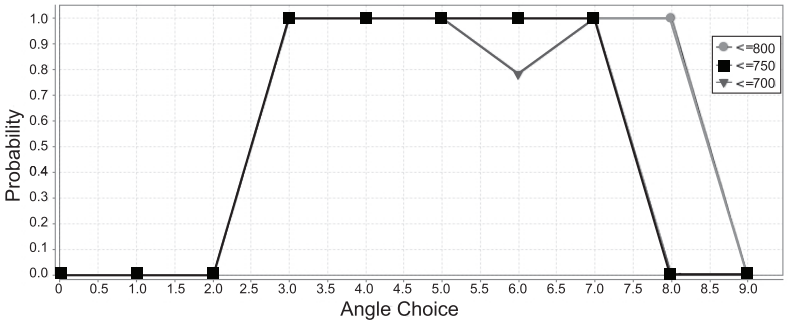
In environment (a) 70% of obstacles require a 70° turn to avoid, and 30% require a 30° turn to avoid. Environment (b) is reverse of the latter, while (c) has 50% of each type of obstacle.

From the results we can derive the most efficient turning angles for each environment type, they are: for (a) a turning angle of 70° is most efficient, for (b) angles of 30°, 40°, 50°, and 70°, and for (c) angles of 30° and 70°.

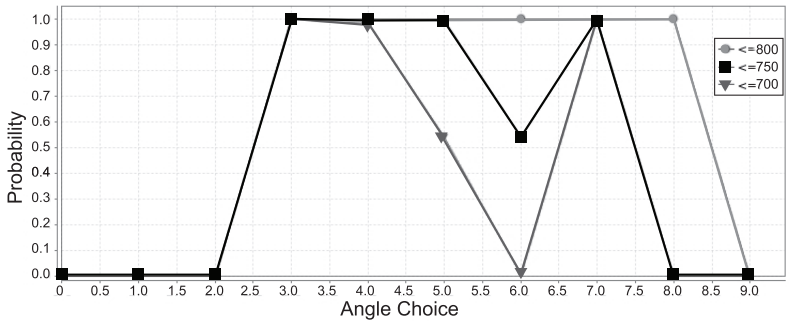
In the next variant of the model the agent learns every time it collides with an obstacle to turn more as a response to the next obstacle encountered. In Figure 3.15 the graphs show the likelihood of the agent learning to choose a certain



(a) More large obstacles.



(b) More small obstacles.

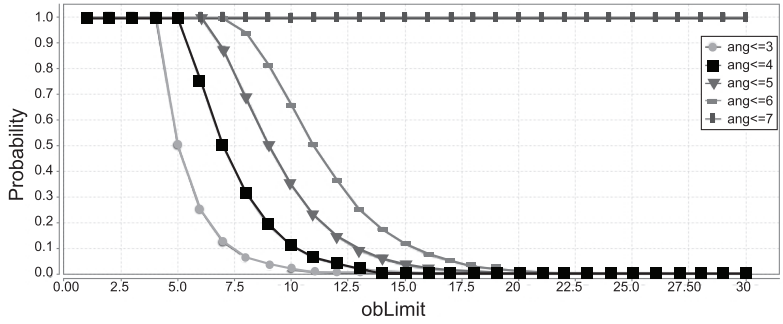


(c) Equal of both obstacles.

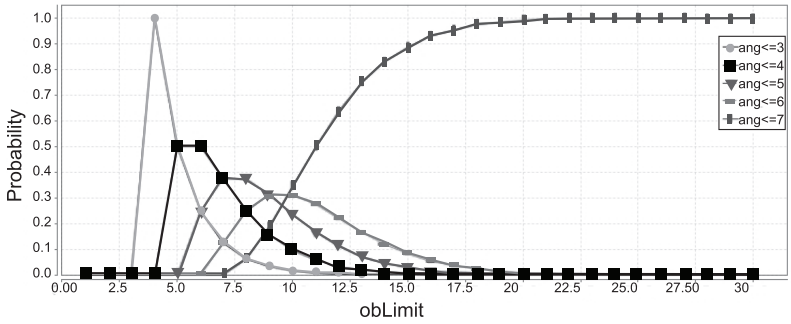
Figure 3.14: Probability of using $<=$ than the energy units: 800, 750, 700.

angle to turn when encountering an obstacle.

The graphs show how as the total number of collisions increase as the agent converges to a turning angle of $70°$. This is to be expected, as this agent is trying to avoid all collisions in this scenario. We derive from this result the number of collisions it takes before an agent settles on its final turning angle. This model's



(a) More large obstacles.



(b) More small obstacles.

Figure 3.15: Probability of choosing $<=$ than the angles: $30°$, $40°$, $50°$, $60°$, $70°$.

code is in Appendix B.4.

Analysis Having a direct comparison of the most efficient turning angle and of the turning angle which a learning agent will finally use, gives us a different way of assessing our ABL systems.

However, there is a disadvantage associated with this PRISM model. In order to have results as probabilities the system needs to be quantified in terms of probabilities. In the case of our systems the probability of an event taking place is commonly unknown –or can only be estimated.

Conversely, if probabilities can be assigned accurately, then the type of quantitative analysis that can be performed is very useful. For example, suppose we know the component failure rates, or their error margins. If the antennas have a 0.1% chance of not registering a contact, this can be expressed in our model. Also,

if the turning motors are only 90% accurate, then when an agent tries to turn x° we can apply a probability that it makes the desired turn or turns a bit too much or little. This type of expressivity is not possible in our PROMELA models.

Chapter 4

Explicit model and simulations

In this section we describe in detail the Explicit model, the system it represents, and the simulations of this system. The Explicit model is based on the ABL systems found in [77]. The system it models differs from the previous systems in Chapter 3, as there is now only one robot. However, it still represents a natural progression in complexity from the preliminary models. This complexity arises from the addition of learning to our model, and from the increase in precision of its representation of the system.

Firstly, we give an overview of the physical system, followed by a detailed description of simulations of this system. We cover the implementation of our model and the results obtained from it. Lastly, we present a comparison between the results of the model and the simulator.

4.1 System model

The type of ABL system we analyse here involves an individual robot that has sets of dual antennas, two proximal (short) and two distal (long). The robot uses ICO learning (see Section 2.2.4) to improve its response to contacting obstacles with its distal antennas, in order to better avoid them.

This robot is situated in a well defined environment that contains a set of obstacles. The possible distributions of obstacles are governed by rules, e.g. en-

vironmental complexity. Environmental complexity is assigned an integer value that governs the minimum possible distance between any two obstacles in an environment. For a full description of the system hardware see Section 2.2.3.

Note that our simulations and models are different to those described in [77] as there are no boundaries imposed on our environments. The removal of boundaries is a simplification that allows us to present a proof of concept analysis with a simpler system. Removing the boundaries also helps us in our abstraction (see Chapter 5). To accommodate the removal of the boundaries we allow the agent to wrap around the environment once it reaches the edge; such that it appears at the opposite side of the environment, in the direction it was facing (the details of the wrapping function are described in Section 4.3). The addition of a formal definition of the minimum distance between obstacles is also a simplification of the systems in [77]; although, restricting this distance does not limit the scope of our approach, as different minimum distances can be chosen for modelling other systems. The minimum distance between obstacles takes into account the wrapping of the environment such that the edge of the environment is considered contiguous with the side that it wraps around to.

The purpose of analysing this ABL system is to assess how well the robot is able to avoid obstacles, and to show how successful the robot is at using learning to navigate its environment.

4.2 Simulations

In this section we describe the simulations carried out for this system. The simulations were done in MATLAB (MATLAB, 2010). The use of simulation is the standard methodology for analysing this type of system. Here, we describe the simulation code and the results obtained by simulation.

The simulator generates a variety of environments, each environment differing in their distribution of obstacles. The robot is represented as an agent that is measured in pixels. An example of the geometry of a simulation is shown in Figure 4.1.

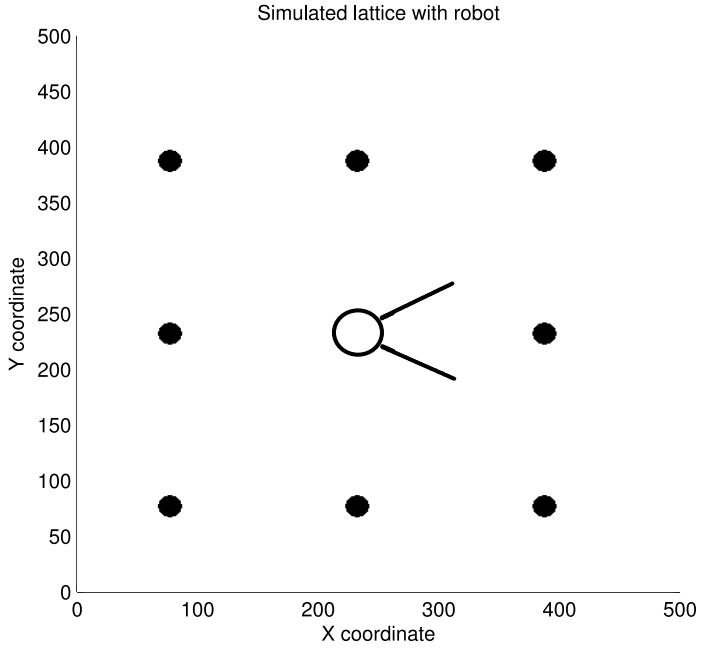


Figure 4.1: Example of the simulation set-up.

In a simulation, the agent is positioned at the coordinates $r_x\,\texttt{(t)}\,, r_y\,\texttt{(t)}$. At each time-step, the agent moves forward one pixel (hypotenuse) at an angle θ (degrees from North); where the change in coordinates is represented by the following equations:

$$r_x\,\texttt{(t)} \;=\; r_x\,\texttt{(t-1)} \;+\; \sin(\theta) \tag{4.1}$$

$$r_y\,\texttt{(t)} \;=\; r_y\,\texttt{(t-1)} \;+\; \cos(\theta) \tag{4.2}$$

The variables in the simulations are stored as floating points. Their coordinates are rounded to integer values when determining whether the agent has collided, but the floating point precision is maintained for movement and learning calculations. The steering angle of the agent is v and is added to θ every time-step; i.e., $\theta\,\texttt{(t+1)} = \theta(t) + v$. The steering angle is generated from the combination of the signals from the antennas; where the signals are combined as in Equation 2.1, from Section 2.2.3. Note that the turning angle can be positive or negative, result-

ing in the agent turning clockwise or anticlockwise respectively.

A set of simulation runs were performed using this setup. The results from these are shown in Figure 4.2.

Figure 4.2 shows the plots of the total number of impacts on distal and proximal antennas over time. The graphs show that, initially, the number of distal and proximal events are close together; this is because the agent has not learned to respond to its distal antennas yet. Over time, the number of proximal events stops increasing as the agent begins to respond to its distal antennas, and hence begins to avoid colliding with its proximal antennas.

Five of the graphs in Figure 4.2 show the predicted behaviour: they show that the agent initially collides with its proximal antennas until it learns to avoid obstacles with its distal antennas. However, with graph F it is unclear as to whether the number of proximal events actually stop increasing. In addition to this, the number of distal events continues to rise, suggesting that the agent is continually contacting obstacles. While this is not problematic for this system, it is unusual. All the agents in the other simulations eventually manage to find a clear path through their environments.

In Section 4.3 we present a PROMELA model representation of this setup, and results from verifying various properties. As we explain in Section 4.4, our results led us to believe that the setup for the simulation illustrated in Figure 4.2.F would eventually stabilise.

4.3 Explicit model

The Explicit model is described in this section. First, we present an overview of the model; next we cover the assumptions made about the system, and then describe the model's PROMELA code and associated functions. We conclude with a detailed analysis of the verifications applied to the model and the results obtained from them.

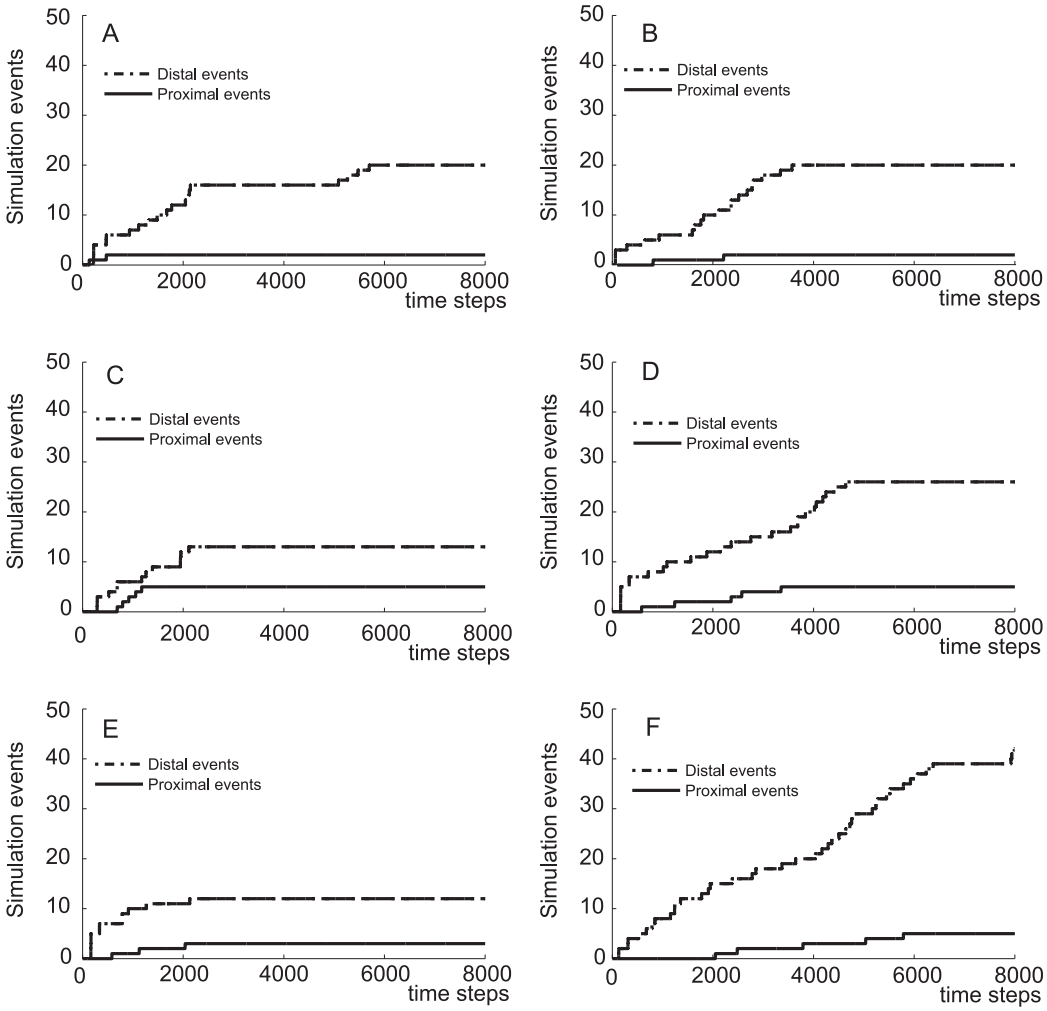


Figure 4.2: Simulation runs for a range of starting directions. A 0°, B 15°, C 30°, D 40°, E 50°, F 58°.

4.3.1 Overview

The Explicit model resembles the system's simulation code as closely as possible. This allows us to be confident about the accuracy of the model. Note that even though the code of the model and simulator may be similar, the model allows us to reason about all possible paths; while the simulator is restricted to only one path at a time. We discuss this in detail in Section 7.5.

We represent the environment as a polar gird; where the centre of the environment is the pole, and angles are measured clockwise from a ray projected North from the pole (this represents the polar axis). The agent, environment, and each obstacle have their centre points stored as polar coordinates. This representation allows for more precise angles to be stored, which is important for these ABL systems because the robots turn an exact angle to avoid obstacles.

As with the simulations, this model has one agent in an open environment; where the agent is learning to use its distal antennas to avoid colliding into obstacles. However, in the Explicit model we use a circular area for the environment. This makes our polar representation less complex for calculating the edges of the environment. Although we use a circular environment for these models, the use of a square, or another shape, for the environment is within the scope of our approach (we address this in Chapter 7).

Figure 4.3 shows an example of an agent in an explicit environment. Here, there are fewer obstacles than in the simulation example (see Figure 4.1), but it is important to note that they are both governed by the same environmental complexity. Given an environmental complexity, it is possible to represent a variety of different configurations of obstacles.

4.3.2 Assumptions

The assumptions we make for this model are the same as those made for the simulator, they are the following. Antennas are assumed to behave as perfect springs, and to only be able to sense one obstacle on a pair of proximal and distal antennas at a time. If there is more than one obstacle along the line of an antenna,

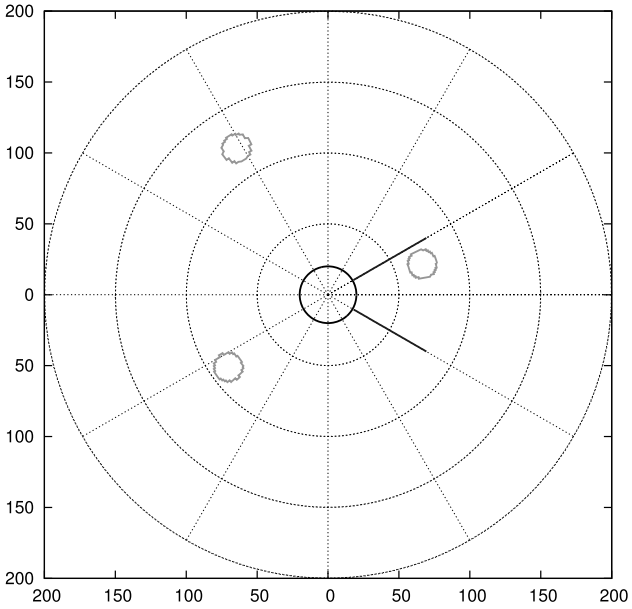


Figure 4.3: Example of an agent in an environment in the Explicit model.

then the closest obstacle is used to produce the signal to the agent. An obstacle is anything in the environment other than free space. The agent starts in the centre of an environment, and there are no obstacles to inhibit it from doing so.

4.3.3 PROMELA code

In this section we give a reduced version of the code used to model this system. First, we give an overview of the code followed by a table of all the inline macro functions (see Section 2.4.1 for inline macros). Next, we describe the PROMELA code, and explain how it is altered for each Explicit model.

Overview

In order to maintain accuracy in the model, we use geometric calculations in embedded `C` code. This requires the setup of some `c_track` (see Section 2.4.1) variables, which are used to monitor the relevant values in the calculations –including

them as part of the state-space.[8]

Each Explicit model contains a set of obstacles stored in an array (this is named `arrObs[OBMAX]` in Figure 4.4). The number of obstacles in the array varies depending on the size and complexity of the model. In order for models to be generated faster, we automate the calculations for a set of legal obstacles. This allows us to generate a random environment for every model. This auto-generation code requires as input: an environmental complexity, a size of obstacle, and a size of environment. From these values, the auto-generation code produces a random array of legal obstacles. Note, for these models we stipulate that there are no obstacles in the centre of the environment, hence we use this as the robot's default starting position. This need not be a fixed restriction and starting point, but it is pragmatic to make sure each model begins without the agent on top of an obstacle.

The main body of the code is in the process `robot()`, which defines the behaviour of the agent. Here, the agent is running an infinite loop where it either continues sensing and trying to avoid obstacles, or it has reached the edge of the environment and is wrapped (`WRAP`) to the other side. Each of the inline macro functions used in this model are described in Table 4.1.

In Figure 4.4 we present a reduced version of an Explicit model's PROMELA code. The process `robot()` uses a `d_step` statement to combine transitions. This better emulates the continuous movement and assessment of the robot in the physical system, where the robot is moving, learning, and turning at the same time.

If `doWrap` is 0, then the agent has not reached the edge of the environment and continues to drive forward while responding to signals from its antennas to avoid obstacles. If `doWrap` is 1, then the agent has reached the edge of the environment and the `WRAP` function is called.

There is a special inline macro function to deal with the situation when an agent collides with an obstacle head on, the `HEAD_ON` function. This type of collision is unique because it means that the agent has crashed into an obstacle

[8]Note that `c_track` variables do not need to be considered as part of the state-space as they can be declared with the parameter "`Unchecked`" which stops them contributing to the state-space. However, they will still be stored on the search stack during verification.

Name	Purpose
SCAN_APPROACHING_OBS	scans the area in front of the robot for obstacles. This area is restricted to distances and angles at which an obstacle may interact with the robot. Uses function GET_OB_REL_TO_ROBOT.
GET_OB_REL_TO_ROBOT	calculates the centre of an obstacle relative to the centre of the robot.
RESPOND	updates the signal from the robot's antennas then calls the RESPOND_TO_OB_BY_TURNING function.
RESPOND_TO_OB_BY_TURNING	turns the robot in response to the signals from its antennas. If the signals indicate a proximal reaction then the LEARN function is called. If the obstacle is touching the robot then the CRASH function is called.
LEARN	causes the robot to learn; i.e., increments ω_d.
CRASH	evaluates the movement of the robot after it has collided with an obstacle. If collision is head-on then the HEAD_ON function is called. Otherwise a proximal turning response occurs.
HEAD_ON	evaluates the movement of the robot after it has collided head-on with an obstacle. Eventually results in a proximal turning response.
MOVE_ROBOT	moves the robot forward, in the direction of its current orientation. Calls the MOVE_FORWARD function.
MOVE_FORWARD	calculates the new position of the robot after moving forward. If the robot has reached the perimeter of the environment, sets a variable (doWrap) to 1.
WRAP	wraps the position of the robot to the other side of the environment, using the point at which the robot approaches the perimeter of the environment and the orientation of the robot as it approaches.

Table 4.1: Inline functions

```
/*Explicit Model: Using Functions (Macros)*/

c_decl { #include <math.h> }
#include "exMoInLines.txt"
#define OBMAX 2

/*Define a polar coordinate to be a distance and angle from origin (pole)*/
/*(Pole: centre of the environment. Polar axis: vertical line directed north.)*/
typedef polarCoord {int d; int a};

/*Setting the C_Track variables*/
/*C_track statements keep track of our C globals.*/
c_track"&x" "sizeof(double)"
/*...etc*/

/*Array of obstacles in the fixed environment*/
polarCoord arrObs[OBMAX];

/*Robot is initialized in the centre of the environment*/
int roboAng, enviDist, enviAng, omegaD, sig, prevSig =0;
byte doWrap, headOn = 0;

proctype robot() {
     do
     :: (doWrap==0) -> d_step{ SCAN_APPROACHING_OBS();
                                    RESPOND();
                                    MOVE_ROBOT();
                                    HEAD_ON();
                                   };
     :: (doWrap==1) -> d_step{ WRAP() };
    od;
};

init {
     d_step{
    /* Set up the polar coordinates of the obstacles - fixed for model */
      arrObs[0].d = 45;    arrObs[0].a = 350;
      arrObs[1].d = 154;    arrObs[1].a = 83;
      };
     atomic{ run robot()};
};
```

Figure 4.4: PROMELA code for the Explicit model

without contacting any of its antennas. In this case the agent continues to try to drive forward, and this results in it slipping to one side of the obstacle, which causes contact with one of its proximal antennas. Because it is largely random as to which side the robot slips to, we select which agent's antenna is contacted nondeterministically. Note that there is a small additional assumption here: that the agent slides to one side of the obstacle. This is a result of the surfaces of the obstacle and the robot being rounded and the fluctuations in the wheels' driving forces. (This behaviour can be observed in the physical system.)

If there is contact with the agent's proximal antennas then the `LEARN()` function is called. The `LEARN()` function implements the ICO learning from Section 2.2.4. If there is a correlation between distal and proximal signals then this causes the agent to respond more vigorously to signals from its distal sensors.

Specifically, the function compares the previous antennas' signals (`Prev_x`) with the current signals (`x`) to determine whether to increment the learning weight (ω_d) of the agent. When model checking, different learning weights represent different states in the system as the agent exhibits different behaviours for different weights.

The `init` declaration is the setup procedure for the model. In here the array of obstacles is initialised with the coordinates of the centre points of all the obstacles, and the agent's process is started.

The functions called in the PROMELA are declared in the underlying `C` code where real variables are used in their calculations. It should be noted that these real variables are maintained and stored throughout verification of the model. In order for some of these values to generate different states in the model their values are rounded to whole numbers and stored separately as part of the state vector.

4.3.4 Verification

As with the simulations, we are concerned with whether the agents manage to learn to avoid obstacles. To assess this we define two properties using LTL formulas. The properties are as follows.

- `1P`: That the signal produced from the proximal antennas will eventually stay zero, indicating that the agent is now only using its distal antennas.
- `2P`: That the learning value eventually stabilises, indicating that the agent has finished learning –stopped crashing.

These properties are formally defined in the following formulas; where `1P` is represented by Formula 4.3, and `2P` is represented by Formula 4.4.

$$\texttt{<>}\ \ \texttt{[]}(\ \ (\texttt{x}\ \texttt{>=}\ 6)\ \ \texttt{||}\ \ (\texttt{x}\ \texttt{<=}\ -6)) \tag{4.3}$$

$$\texttt{<>}\ \ \texttt{[]}\ \ (\omega_d\ \texttt{<=}\ \texttt{Max\_}\omega_d) \tag{4.4}$$

In Formula 4.3, `x` denotes the signal difference between the agent's antennas; it ranges from -6 to 6, where both extremes indicate a proximal reaction. In Formula 4.4, the constant `Max_`ω_d represents the maximum level of learning possible

for a particular model. It is calculated through a series of simpler verifications where it is iteratively narrowed down until its highest possible value is found. It is possible to include an additional state variable in the model that would remove the need for calculating Max_ω_d. That is, the C code variable pLearn could be tracked as a state variable. This becomes equal to 1 whenever learning occurs, and to 0 after learning. By using this we can define the alternate Formula 4.5, which represents the property that it will eventually, always be true that pLearn remains equal to 0; hence, that learning stabilises (stops).

```
<> [] ( pLearn == 0 )
```
(4.5)

The drawback of tracking pLearn as a state variable is that it will increase the state-space of the model. For verifying these Explicit models it was straightforward to get the verifications without it and we were also interested in the value of Max_ω_d for a given model. However, if we are no longer interested in Max_ω_d and are to increase the size and number of environments for the models, then using pLearn in this way would speed up the verification process.

Property 1P was checked and shown to be false. Initially, it was suggested that this indicated that the agent was unable to learn to avoid colliding with obstacles. However, from this failed verification we examined a counterexample, which highlighted a path in the model that caused the verification to fail. It showed that the property was not violated because of the agent being unable to learn to avoid obstacles that it could sense, but was violated because the agent was always able to collide, head on, with obstacles that it could not sense. That is to say: there was sufficient space between the agent's antennas for an obstacle to pass, undetected. The agent's ICO learning, meant that this type of collision did not trigger it to learn, as ICO learning requires the presence of a previous distal signal to correlate the proximal signal with. This initial check allowed us to redefine property 1P, so that it related to the learning of the agent. It also prompted discussion about the effectiveness of the robot's parameters in the physical system.

We redefined property 1P such that it better reflects the learning of the agent. Hence, the property is only violated if there is a proximal reaction that was pre-

ceded by a distal reaction. We now checked whether it is eventually, always true that a proximal reaction implies that there was no previous distal reaction. This new version of the property is expressed in Formula 4.6.

$$\texttt{<> [](\ p\ \rightarrow\ !d\ )} \tag{4.6}$$

Here, we define `p` as `((x <= -6)||(x >= 6))` (proximal signal), and `d` is defined as `((Prev_x != 0)&&(Prev_x > -6)&&(Prev_x < 6))` (distal signal). This property is shown to be true for our set of example environments.

The properties `1P` (as Formula 4.6) and `2P` were verified on six automatically generated environments. Each model has the same environmental complexity, but a different distribution of obstacles. The six environments used for the verifications are shown in Figure 4.5. These are the starting locations of the models. The agent is in the centre facing East, and the obstacles are shown as hollow, grey circles.

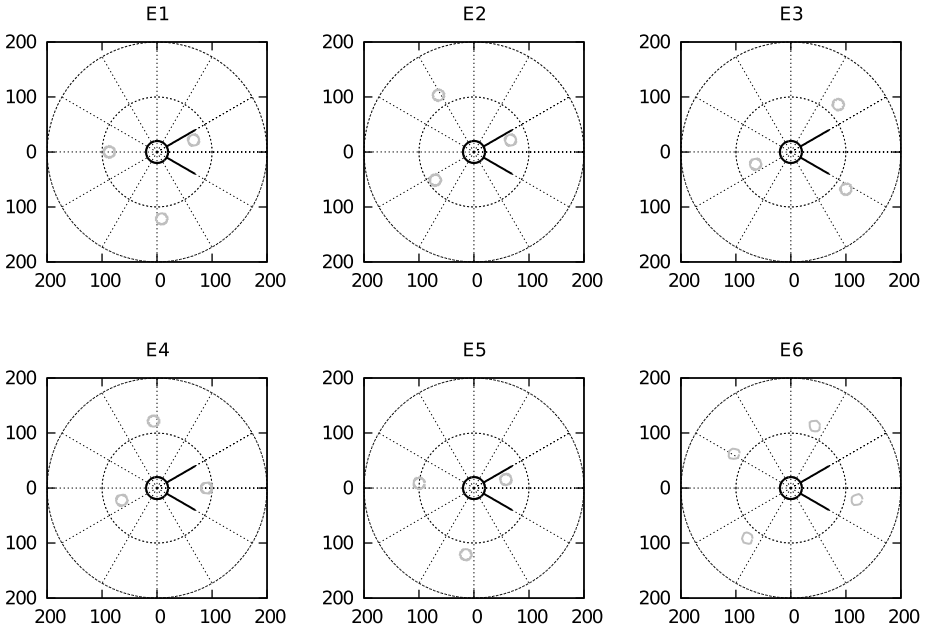


Figure 4.5: Environments E1 - E6.

The results of the verifications for the models in Figure 4.5 are presented in Table 4.2. Here, the environment's labels, `E1` $\rightarrow$ `E6`, correspond to those in Fig-

Environment	Property	Max_ω_d	Stored states	Max search depth	Time (sec)
E1	1P	1	273664	547323	2.19
	2P		273734	547323	2.26
E2	1P	1	150562	300979	1.03
	2P		150492	300979	1.04
E3	1P	0	670	1339	0.00
	2P		670	1339	0.00
E4	1P	0	77034	154067	0.16
	2P		77034	154067	0.28
E5	1P	1	218326	436647	1.36
	2P		218372	436647	1.73
E6	1P	1	61256	122507	0.28
	2P		61434	122507	0.52

Table 4.2: Verification results for the Explicit model

ure 4.5. `Max_`ω_d is calculated separately for each environment. Note that in environments `E3` and `E4` the maximum learning possible is actually zero. This is not because the agents don't interact with the obstacles, but because those interactions don't involve contact with their distal antennas. The prior contact with the distal antennas is the key component of ICO learning; therefore, without it no learning takes place.

In `E3` and `E4`, the agents have continuous proximal collisions with an obstacle; they turn 90° on each impact, and then repeat the proximal collision from a different angle. Hence, these models get locked into repetitive sequences of proximal collisions and therefore exhibit no variance in behaviour (no learning). As a consequence, the state-spaces for these models are much lower than the others. This is an interesting observation for an individual environment; though, as the number of obstacles is increased then the likelihood of this decreases. In fact, for this type of environment, this scenario is not useful if learning is to be assessed.

Table 4.2 also shows the total states stored when running the verifications. In `E3` and `E4`, the number of *Stored states* are low relative to the other models because of the lack of learning and distal reactions in these models. The *Max search depth* is the length of the longest path explored when verifying the property. On the right of the table the time taken to run each verification is displayed in seconds. Properties `1P` and `2P` were successfully verified for each model.

4.4 Comparison and analysis

In this section we present a comparison between the simulations and the verifications of the Explicit model for analysing this ABL system.

The trends observed in the simulations were backed up by the verification results from the Explicit model. Although, as noted in Section 4.2, the simulations were unable to make any definitive statements about whether the agents would eventually stop learning, once they began to learn to avoid obstacles. What the Explicit model provides are definitive statements.

Figure 4.6 shows the extended run of the simulation setup of graph `F`, from Figure 4.2. After running verifications with the Explicit model, we showed that this simulation setup should eventually stabilise. Hence, there should be no more distal or proximal events, which was not shown in the simulation. This prompted us to query the results shown in graph `F`, and in doing so we ran an extended simulation to see if, indeed, the system did eventually stabilise. This extended run shows that the agent's distal and proximal events do eventually stabilise, as our Explicit model predicted. This demonstrates a benefit from applying model checking alongside simulation for ABL system analysis.

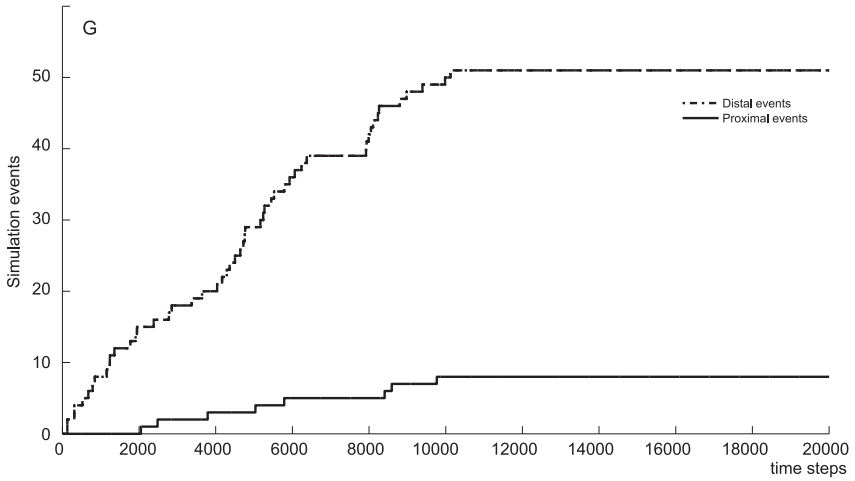


Figure 4.6: Simulation run starting direction `G` $58°$ with extended running time.

The use of simulation is cheap: straightforward to implement and run. Ad-

ditionally, it provides a reasonable assessment of the trends for the behaviour of agents in a given environment. The use of model checking however, allows for more definitive statements to be made, and when used in unison with simulation can help guide the simulations. This guidance is exemplified by the benefit of the extended run in Figure 4.6. Of course, in a different simulation run perhaps the properties that were verified by model checking could have been shown. It is impossible to be sure though. Running simulations for as long as is necessary also poses a problem because it is not always apparent from the system's specification as to how long it will take to stabilise. For example, the batch of simulations from Figure 4.2 took approximately 60 minutes total run time. Additionally, the extended run was required that took a further 25 minutes. The benefits of model checking here are that it is much faster (a matter of seconds per verification, see Tables 4.2 and 6.1) and it never misses an error path. Although, of course, it is the definition of the property being checked that determines what an error path is.

Having to generate a different model for each environment greatly increases the amount of time it takes to model a type of system. While it can provide some specific analysis of individual environments, having a way to model different examples of a specific type of system in one model would provide a means for much more efficient and thorough analysis. This is the type of model that we can generate with our *Agent-centric abstraction*. We describe this abstraction methodology in Chapter 5.

Chapter 5

Agent-centric abstraction

In this chapter we describe our abstraction methodology, and present a proof of its correctness. We begin with an overview of the Agent-centric abstraction, and following this we define the scope of the abstraction by explaining the parameters and calculations involved in generating it. We formally define all the components and functions that are required to prove the correctness of the abstraction, and then present its proof.

5.1 Overview

The previous models that we have presented deal with a certain type of ABL system and focus on specific instances of it. By focusing on a specific agent, environment, and type of learning, we can formally reason about this system's properties, but not about all instances of this type of system. Here, we propose something with a larger scope. The idea behind Agent-centric abstraction is to be able to reason about an entire class of ABL system in one model; as opposed to creating an Explicit model for each variation of the type.

In order to achieve our Agent-centric abstraction, we have to make some assumptions about the type of system that we are dealing with. It is our experience gained via the Explicit model (see Section 4.3) that informs these assumptions. Our assumptions are deliberately strict –restricting us to systems involving a sin-

gle robot and a given complexity of environment. This simplification allows us to describe our approach more clearly, and to prove its soundness more succinctly. Our approach can be extended to more elaborate systems. Some examples are discussed in Chapter 7.

Agent-centric abstraction essentially involves merging states from different (Explicit) models into a single state in a model called the Relative model. These different states are associated with different environments, and different positions of the robot, but are equivalent from the perspective of the robot. This abstracted model is referred to as the Relative model because it represents the system relative to the perspective of the robot.

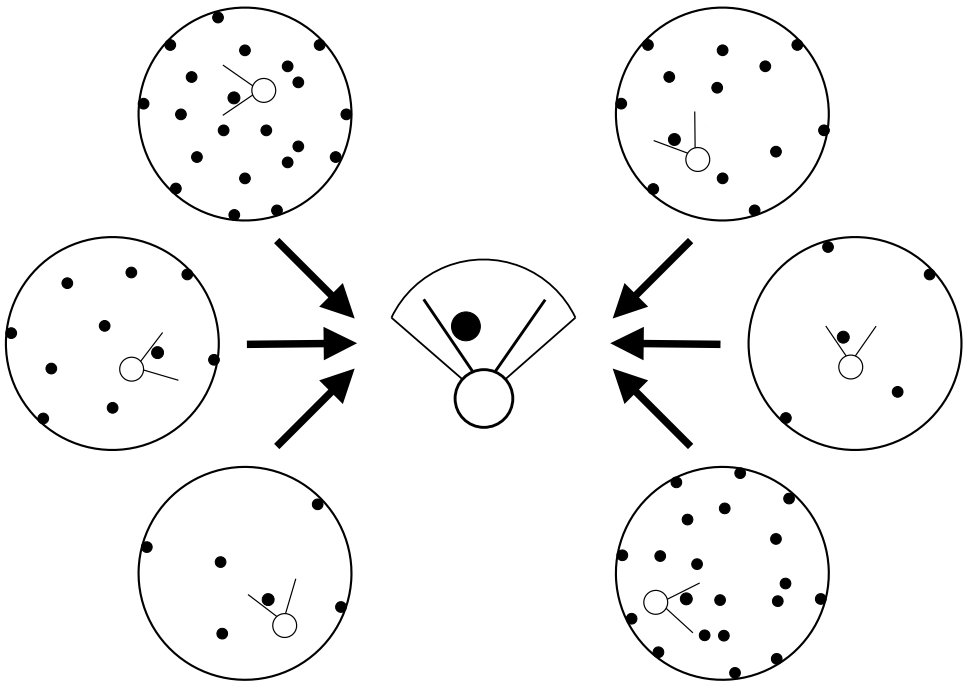


Figure 5.1: States from Explicit models merged into the *Cone of influence* representation (centre) via Agent-centric abstraction.

Figure 5.1 illustrates the main concept of Agent-centric abstraction. Multiple explicit environments are mapped to the *Cone Of Influence* (COI) representation of the system. The COI representation can be visualised here in the centre of the figure (it is defined in Section 5.2.3).

In the Relative model, we only consider the position of any obstacle relative to the agent at any state; i.e., we only consider the presence or otherwise of an obstacle within a conical area in front of the robot (within the COI). This information is all that is required to determine the next movement of the agent. Because of the equivalence of states that are merged in the Relative model, any property that holds for it also holds for any Explicit model that satisfies our assumptions for the system. We will describe an instantiation of the Relative model in Chapter 6. Of course, using a single abstracted model (rather than an entire class of models) to verify our properties is far more efficient. However, we must prove that the equivalence holds, and that this implies the preservation of properties. We do this in Section 5.5.

In order to show the value of our approach we provide a proof that there exists a modified simulation relation between the Explicit and Relative models of the system, such that the Relative model simulates the Explicit model.

5.2 Assumptions

The assumptions for the abstraction are informed by our verifications with the preliminary and Explicit models, they are as follows.

- `A1`: No two obstacles can be in the COI at any time.
- `A2`: No obstacle can cause the robot to learn via a collision that does not involve contact with the robot's distal sensors.
- `A3`: No turn and movement sequence (within one time-step in the system), by the robot, to avoid one obstacle can cause an immediate collision into another obstacle.

The parameters that are used in these calculations are the: diameters of the robot and obstacles, length of and angle between antennas, maximum turn and movement of the robot (within one time-step in the system), and environmental complexity (see Section 4.1). We calculate the constraints for assumptions `A2` and `A3` below.

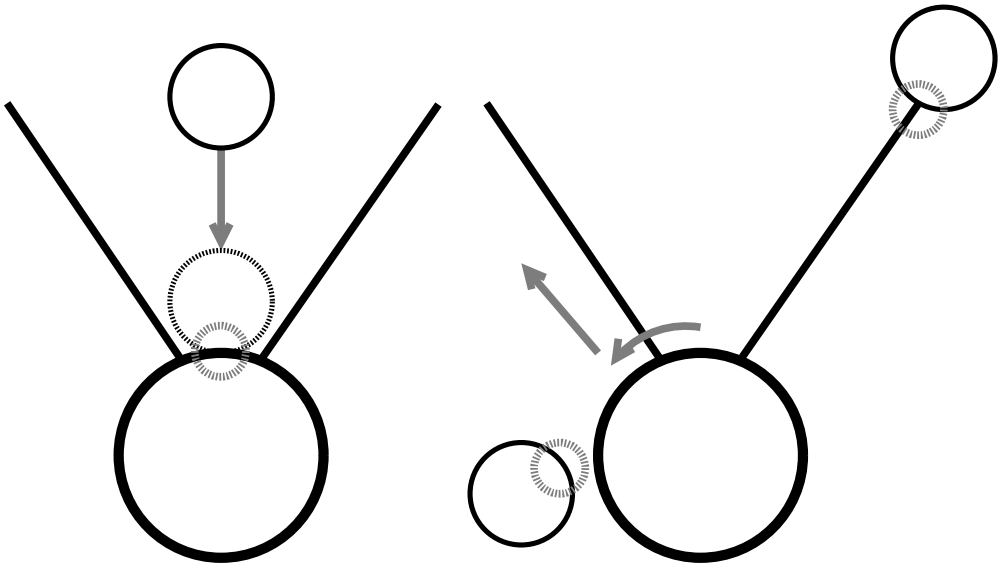


Figure 5.2: (a) Direct collision without contacting antennas. (b) Indirect collision without contacting antennas.

Two of the main situations that can lead to a violation of these assumptions are as follows. One, shown in Figure 5.2(a), where an obstacle collides directly into the agent from the front, we refer to this as a *direct collision*. Two, shown in Figure 5.2(b), where an agent turns to avoid one obstacle and collides into another without it first contacting an antenna, we refer to this as an *indirect collision*. To assess whether the first situation is possible requires a straightforward calculation while to assess the latter requires a more convoluted one.

Note that for the situation described in `A3` to occur, an additional assumption is inferred: that an obstacle can get into a position behind the robot's antenna that makes the indirect collision possible. For our calculations we make this assumption, as a *worst case scenario*. That is, we assume that an obstacle can end up in this position, and calculate whether it is then possible for a indirect collision to occur.

An Explicit model can be used to discover whether it is, in fact, possible to manoeuvre an obstacle to such a position for a given environment. In the Agent-centric abstraction, however, we are concerned with all variations of a type of

environment. Therefore, we assume the worst case scenario is possible. As a consequence of this additional assumption, even if our calculations show that `A3` is false, it is still possible that the obstacle could never have been in a position to cause the collision in the first place. Hence, these calculations simply act as suggestions as to what should be represented in the model.

5.2.1 Direct collision

In Figure 5.2(a), a direct collision occurs when an obstacle passes between a robot's antennas. This is possible when the combination of the angle between antennas and diameter of an obstacle allow the obstacle between the antennas without contacting them.

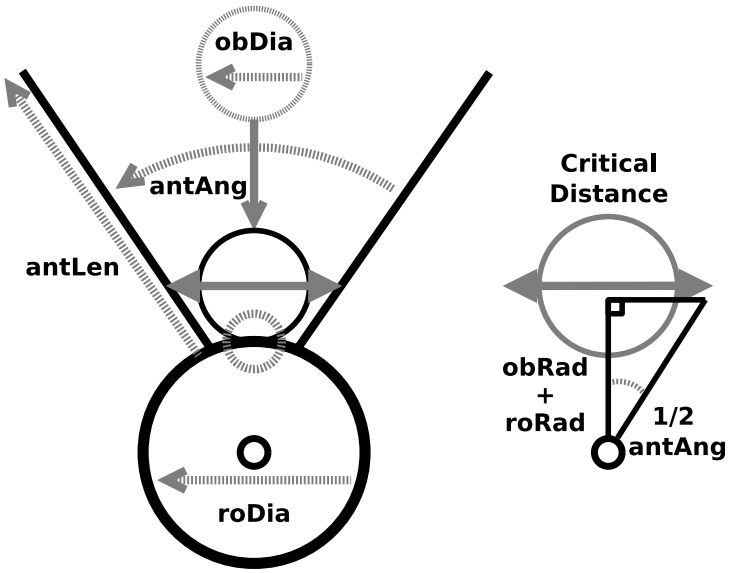


Figure 5.3: Measurements for a direct collision, without contacting antennas.

Figure 5.3 illustrates the *Critical distance* which is calculated to determine whether an obstacle can fit between the antennas without contact. The relevant parameters for this calculation are as follows: `antLen`, the length of an antenna; `obDia`, the diameter of an obstacle; `obRad`, the radius of an obstacle; `antAng`, the angular distance between the antennas; `roDia`, the diameter of the robot; and

roRad, the radius of the robot. The *Critical distance* and whether it leads to a *Direct collision* are calculated in Equations 5.1.

$$\begin{aligned} \textit{Critical distance} &= 2 * (\tan(0.5 * \texttt{antAng}) * (\texttt{obRad} + \texttt{roRad})) \\ \textit{Direct collision} &= \begin{cases} \texttt{true}, & \texttt{if}\,(\textit{Critical Distance} > \texttt{obDia}) \\ \texttt{false}, & \texttt{otherwise} \end{cases} \end{aligned} \tag{5.1}$$

Indefinite proximal reactions

In addition to the calculation of *Critical distance*, it is necessary to calculate the possibility of *Indefinite Proximal Reactions* (IPRs). This is the situation in which the obstacle cannot pass between the antennas without contact, but only the proximal antennas are contacted. If IPRs are possible then the robot can continuously learn avoidance behaviour without being able to apply it within its environment. However, this is only the case when a proximal reaction is sufficient to trigger a learning response.

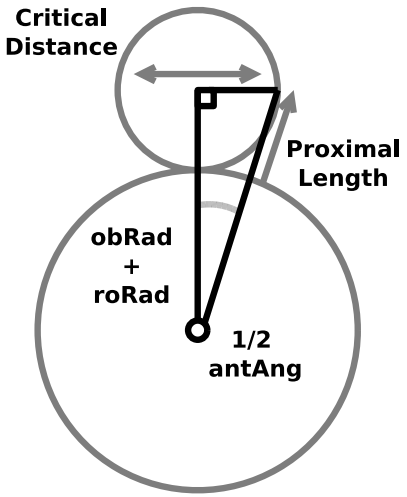


Figure 5.4: Identifying indefinite proximal reactions from a direct collision scenario.

Figure 5.4 shows the measurements taken for this calculation. The value of

Maximum proximal length (MPL) and whether it leads to an IPR are calculated in Equations 5.2.

$$\begin{aligned} MPL &= (\texttt{obRad} \div (\sin(0.5 * \texttt{antAng}))) - \texttt{roRad} \\ IPR &= \begin{cases} \texttt{true}, & \texttt{if}\,(\texttt{proximal length} \geq MPL) \\ \texttt{false}, & \texttt{otherwise} \end{cases} \end{aligned} \tag{5.2}$$

Even where a correlation between the distal and proximal antennas is needed to trigger learning, the possibility of continuous learning can occur. For example, suppose a distal reaction to one obstacle causes the robot to have direct collision with its proximal antennas with another obstacle. The robot's response to the first obstacle may have been optimal avoidance behaviour, but as a result of the consequent and unavoidable proximal reaction, the agent will incorrectly learn to respond more vigorously to its distal antennas. For our models, however, this situation violates assumption `A1`.

5.2.2 Indirect collisions

These calculations involve checking how far apart obstacles can be while still causing an indirect collision; i.e., what is the least complex environment that still allows an indirect collision to occur.

Figure 5.5(a) shows the precursor to an indirect collision, where obstacle A passes by the robot's antenna without contact. Then contact is made with obstacle B, causing the robot to turn anticlockwise. The robot then continues to move forward, which causes A to collide with its side (which we refer to as an *indirect collision*).

Figure 5.5(b) highlights the measurements for where A collides with the side of the robot. The lowest position of A, indicated by the label $A2$ and a black centre, is the farthest position that obstacle A can be from obstacle B that still causes an indirect collision. $MaxTurn$ represents the maximum angle that the robot can turn from a contact on its opposite antenna's farthest point (the point

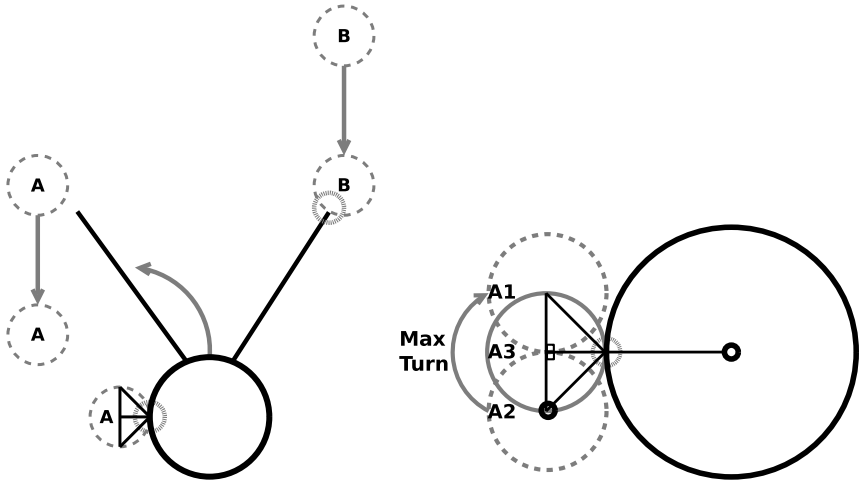


Figure 5.5: (a): Turning response. (b): Obstacle positions in a turning response.

where B contacts the right antenna) that still allows A into a position where an indirect collision is possible.

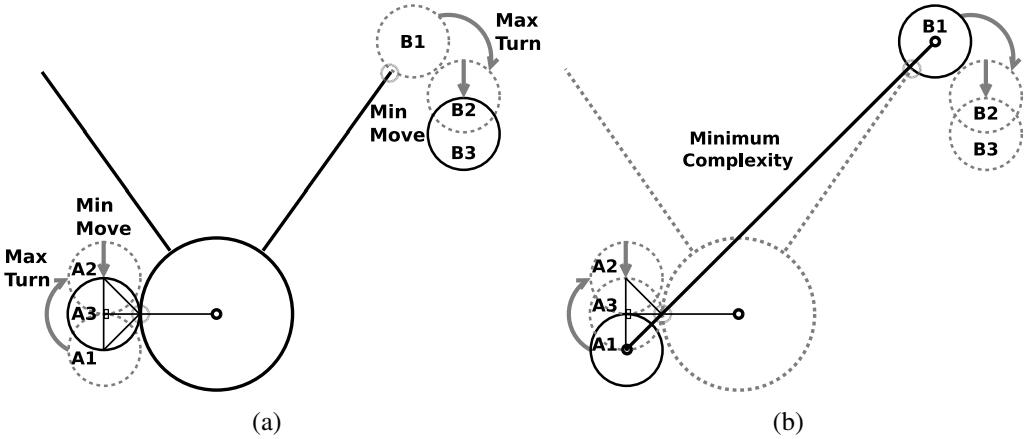


Figure 5.6: (a) Turn and move into indirect collision. (b) Minimum environmental complexity measure.

Figure 5.6(a) shows the robot's turn and subsequent movement in response to sensing obstacle B. Figure 5.6(b) highlights the farthest distance between the two obstacles that can lead to an indirect collision: shows the minimum environmental complexity (labelled *Minimum Complexity*) which could lead to an indirect collision. To calculate this distance we add, the: diameter of the robot (`roDia`),

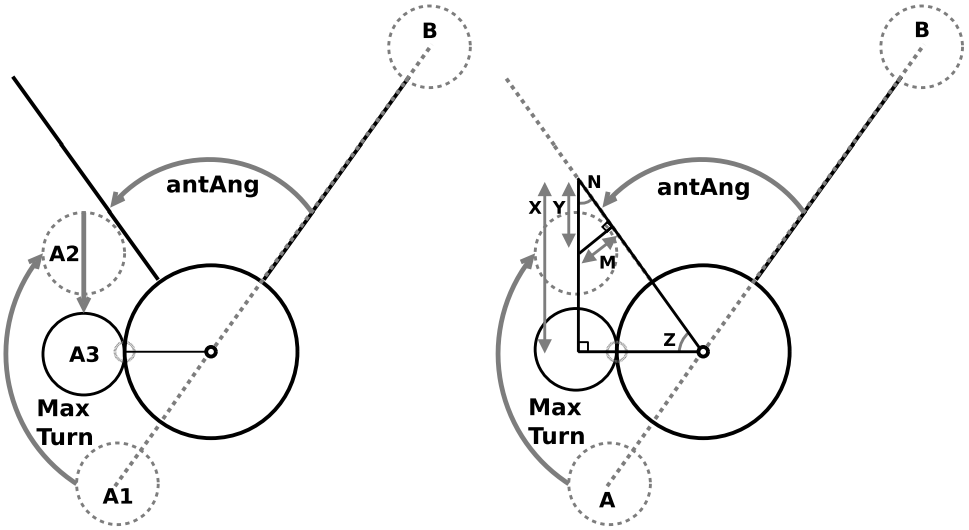


Figure 5.7: (a) Maximum turn, and distance between obstacles. (b) Geometric calculations for pre-turn obstacle.

length of its antenna (`antLen`), radius of both obstacles (`obDia`), and distance that obstacle A is from behind the robot (at position $A1$) before the turn to the final collision takes place.

In Figure 5.7(a), for A to collide with the robot it has to move from position $A2$ to position $A3$. This measurement is represented by `X` in Figure 5.7(b). The position $A1$ is the farthest that obstacle A can be from the robot while still being able to move onto line `X`. Here, A can get as close to the antenna as possible, but without touching it; hence, we can calculate M as the radius of an obstacle plus the minimum unit of distance from an antenna that does not result in detection. We use M to calculate the distance `Y`, where if `Y` is greater than `X` then the obstacle is already in an invalid position (possibly overlapping with the robot), and therefore there is no *Possible Indirect Collision* (PIC) for this system. These calculations are shown in Equations 5.3.

$$
\begin{aligned}
\texttt{Z} &= 90 - (\texttt{antAng} \div 2) \\
\texttt{X} &= (\texttt{obRad} + \texttt{roRad}) * \tan(\texttt{Z}) \\
\texttt{M} &= \texttt{roRad} + 1 \\
\texttt{N} &= \texttt{antAng} \div 2 \\
\texttt{Y} &= \texttt{M} \div (\sin(\texttt{N}))
\end{aligned}
\tag{5.3}
$$

$$
\texttt{PIC} = \begin{cases} \texttt{false}, & \texttt{if}\,(\texttt{Y} \geq \texttt{X}) \\ \texttt{true}, & \texttt{otherwise} \end{cases}
$$

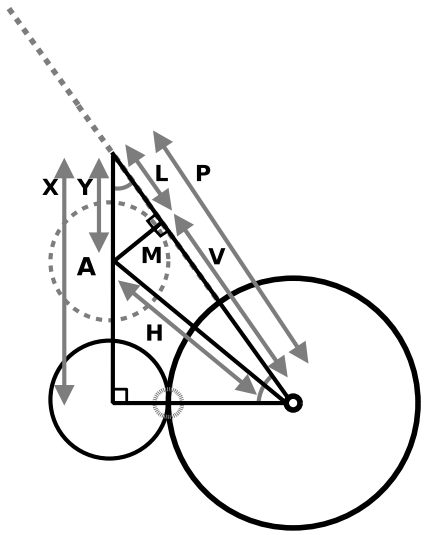


Figure 5.8: Identifying indefinite proximal reactions from an indirect collision scenario.

Next we calculate the distance between A and the robot. This measurement is represented by `H` in Figure 5.8. It represents the distance from the centre of A to the centre of the robot. Since `H` remains the same when the robot turns, we use it in Equations 5.4 to calculate the *Maximum Distance Between Obstacles* (MDBO) –this is between the centre points of obstacles.

$$
\begin{aligned}
\texttt{P} &= \sqrt{(\texttt{obRad} + \texttt{roRad})^2 + \texttt{X}^2} \\
\texttt{L} &= \sqrt{\texttt{Y}^2 - \texttt{M}^2} \\
\texttt{V} &= \texttt{P - L} \\
\texttt{H} &= \sqrt{\texttt{M}^2 + \texttt{V}^2} \\
\\
\texttt{MDBO} &= \texttt{H} + \texttt{roRad} + \texttt{antLen} + \texttt{obRad}
\end{aligned}
\tag{5.4}
$$

By adjusting the variables it is also possible to calculate whether IPR can occur from this situation. Unlike the situation with IPR for a *direct collision*, this is potentially an issue even if the robot correlates signals to learn. For example, suppose the robot responds to a distal signal by turning directly into another proximal collision. The calculation adjustment is shown in Equations 5.5.

$$
\begin{aligned}
&\cdots \\
&\texttt{M} = \texttt{roRad} \\
&\cdots \\
&\texttt{IPR} = \begin{cases} \texttt{true}, & \texttt{((roRad + obRad) < V < (roRad + ProxLen))} \\ \texttt{false}, & \texttt{otherwise} \end{cases}
\end{aligned}
\tag{5.5}
$$

The only alteration to the formula is with `M`. However, this changes the calculation. Note that if PIC was true before then it does not need to be recalculated, as the value of `Y` will now be lower.

These calculations provide a static check that the assumptions `A2` and `A3` are satisfied. Thus, demonstrating that no direct or indirect collisions can occur, and that our Agent-centric abstraction can be applied (without having to consider these situations). However, if the calculations indicate that these collisions can occur, this is still based on the assumption of the worst case scenario and therefore it only suggests that these situations should be considered in the model, opposed to guaranteeing that they occur.

Note that these calculations are used here to restrict a system to the specific Agent-centric abstraction we present. By applying the same concept of a COI

representation, but adjusting how obstacles are encountered, these situations can be included. For example, by allowing obstacles to enter from the side of the COI we can represent indirect collisions.

5.2.3 Cone of influence

The *Cone Of Influence* (COI) is a cone-shaped area in front of the robot which represents the set of coordinates at which an obstacle can interact with the robot. Its size is calculated from the specification of the robot, environment, and the obstacles within the environment.

For the Agent-centric abstraction we present here, the maximum distance between any two points in the COI is less than the minimum distance between any two obstacles in an environment (environmental complexity); i.e., only one obstacle can be in the COI at once. When obstacles appear in the COI, they do so from the front. This is because the robot is continuously moving forward and that the environmental complexity selected is low enough that no obstacle can enter from the side of the COI (see Section 5.2.2). The COI represents all possible encounters of an obstacle for a given class of system.

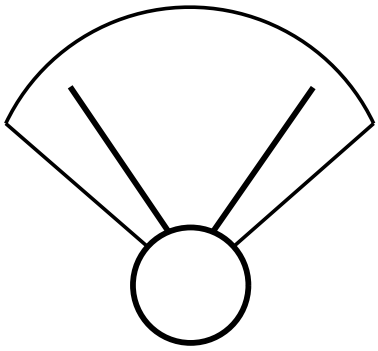


Figure 5.9: Agent-centric abstraction COI representation.

The COI is shown in Figure 5.9. The widest point of the COI is calculated by adding the distance between the tips of the robot's antennas to the radius of two obstacles. Also, the edge of the cone must be at least the width of an obstacle's radius from the robot's antennas.

5.3 Formal definitions

We aim to prove that a simulation relation exists between an Explicit model and the model generated from our Agent-centric abstraction, the Relative model. In order to show this, we must first provide formal definitions of both models and the components of the ABL system that they represent. In the first section, we discuss the notation used in all the subsequent definitions, followed by definitions of each component. We use set-notation in our definitions.

5.3.1 Notation

In this section we define the symbols, models, and special terms that aid the description of our functions. The notation we use is as follows.

- Let $\mathcal{A}$ represent the set of all angles (in degrees).
  Thus $\mathcal{A} = \{\theta \in \mathbb{N}^0 : 0 \leq \theta \leq 359\}$
- Let $\mathcal{D}$ represent the set of possible distances in an Explicit model.
  Thus $\mathcal{D} = \{\eta \in \mathbb{N}^0 : 0 \leq \eta \leq 400\}$
- Let $\mathcal{A}' \subseteq \mathcal{A}$ such that $\mathcal{A}' = \{\theta \in \mathbb{N}^0 : 0 \leq \theta \leq 79\}$
- Let $\mathcal{D}' \subseteq \mathcal{D}$ such that $\mathcal{D}' = \{\eta \in \mathbb{N}^0 : 0 \leq \eta \leq 90\}$
- Let σ represent a set of all possible values of the agent's antennas' signals. Thus $\sigma = \{\eta \in \mathbb{Z} : -6 \geq \eta \leq 6\}$
- Let $\mathcal{C}$ represent a coordinate as a distance-angle pair (d, a); where $d \in \mathcal{D}$ and $a \in \mathcal{A}$
- Let ω_d represent the value which quantifies the agent's learning, (learning weight) such that $\omega_d \in \{\eta \in \mathbb{N}^0 : 0 \leq \eta \leq \omega_{dMAX}\}$, where ω_{dMAX} is assigned to the highest possible value of agent's learning weight.
- Let $\mathcal{S}_L(x)$ represent the set of coordinates of all obstacles' centre-points that are in the COI and would cause that obstacle to touch section x of the *left* antenna, where x ranges from $1 \rightarrow 6$.
- Let $\mathcal{S}_R(x)$ represent the set of coordinates of all obstacles' centre-points that are in the COI and would cause that obstacle to touch section x of the *right* antenna, where x ranges from $1 \rightarrow 6$.

5.3.2 Explicit model definition

The Explicit model is a detailed model of a unique environment containing obstacles and a robot (as described in Section 4.3). A state in this model consists of: the coordinates of the centre point of a robot, its current trajectory in its environment, the signal difference of its antennas, and its learning weight.

A state comprises the following variables: eA, eD, rA, $aDif$, and lW. Angle eA and distance eD represent the polar coordinate (eD,eA) of the agent relative to the centre of the environment. Angle rA is the direction the robot is facing relative to North. The difference between the antenna signals is $aDif$, and lW is robot's learning weight. These variables are bound by the following constraints: $eA \in \mathcal{A}$, $eD \in \mathcal{D}$, $rA \in \mathcal{A}$, $aDif \in \sigma$, and $lW \in \omega_d$.

Definition 5.1. We define an Explicit model as a Kripke structure $\mathcal{M}_E$, where $\mathcal{M}_E = (AP_E, S_E, s_{E0}, R_E, L_E)$. Set AP_E contains all *atomic propositions* (AP) in $\mathcal{M}_E$. Set S_E contains all states s_E, where s_E is a tuple, $s_E = (eA, eD, rA, aDif, lW)$. The initial state $s_{E0} = (0, 0, 0, 0, 0)$, where the robot is in the centre of the environment, facing North. The transition relation $R_E \subseteq S_E \times S_E$. Each element in this relation is a transition from a state s_{En} to s_{En+1}, where $s_{En} = (\overline{eA}, \overline{eD}, \overline{rA}, \overline{aDif}, \overline{lW})$ and $s_{En+1} = (\overline{eA}', \overline{eD}', \overline{rA}', \overline{aDif}', \overline{lW}')$. (The over-line implies an explicit value.) The function $L_E : S_E \rightarrow 2^{AP_E}$, returning the set of AP true at a state.

Assumption

In an Explicit model there are no obstacles within $91units$ from the centre of the environment. This ensures that for both our models the agents start in free-space; i.e., in their initial state they are not in contact with any obstacle.

5.3.3 Relative model definition

The Relative model is an abstraction of the Explicit model, which uses the COI representation of an ABL system (see Section 5.2.3). In order to translate from the Explicit model to the Relative model we need to calculate the COI representation

for a state in the Explicit model.

A state comprises the following variables: $relA$, $relD$, $aDif$, and lW. The variables $aDif$ and lW are the same as with the Explicit model. Angle $relA$ and distance $relD$ represent the polar coordinate ($relD$,$relA$) of the nearest obstacle relative to the centre of the agent. These variables are bound by the following constraints: $relA \in \mathcal{A}'$, and $relD \in \mathcal{D}'$.

Definition 5.2. We define a Relative model as a Kripke structure $\mathcal{M}_R$. $\mathcal{M}_R = (AP_R, S_R, s_{R0}, R_R, L_R)$. Set AP_R contains all AP in $\mathcal{M}_R$. Set S_R contains all states s_R, where s_R is a tuple, $s_R = (relA, relD, aDif, lW)$. The initial state $s_{R0} = (0, 0, 0, 0)$, where the robot is in *free-space*. The transition relation $R_R \subseteq S_R \times S_R$. Each element in this relation is a transition from a state s_{Rn} to s_{Rn+1}, where $s_{Rn} = (\overline{relA}, \overline{relD}, \overline{aDif}, \overline{lW})$ and $s_{Rn+1} = (\overline{relA}', \overline{relD}', \overline{aDif}', \overline{lW}')$. The function $L_R : S_R \rightarrow 2^{AP_R}$, returning the set of AP true at a state.

5.4 Function definitions

Several functions are used to map states to successor states in the Explicit and Relative models. We refer to these as transition functions, and they are $\mathcal{F}_E$ and $\mathcal{F}_R$. Additionally, we have translation functions $\mathcal{T}_1$ and $\mathcal{T}_2$ that are used to translate from a state in one model to a state in the other.

5.4.1 Transition function $\mathcal{F}_E$

Transition functions $\mathcal{F}_E$ and $\mathcal{F}_R$ determine transitions in the Explicit and Relative models respectively. In fact, a transition (s_{En}, s_{En+1}) in the Explicit model involves first projecting the current state s_{En} to a state in which the agent is in the centre (pole) of its COI, executing a transition from this state, then reflecting back into the Explicit model. Specifically the projection is to a state in the Relative model (a state s_{Rn}). So transition function $\mathcal{F}_E$ is composed of the other functions, $\mathcal{T}_1$ (projection in the Relative model), $\mathcal{F}_R$ (transition in the Relative model), and $\mathcal{T}_2$

(reflection back to the Explicit model). This situation is illustrated in Figure 5.10.

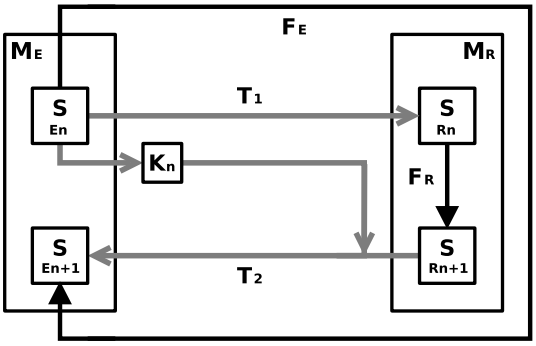


Figure 5.10: Mapping of the transition function $\mathcal{F}_E$.

Note that $\mathcal{K}_n$ stores information about the original state (the position of the agent in its environment) that is necessary for the reflection $\mathcal{T}_2$ back to the Explicit model. We refer to this information as the *key*. Specifically this a triple, $\{eA, eD, rA\}$, where (eA, eD) is the polar coordinate and rA is the facing direction of the robot in the Explicit model.

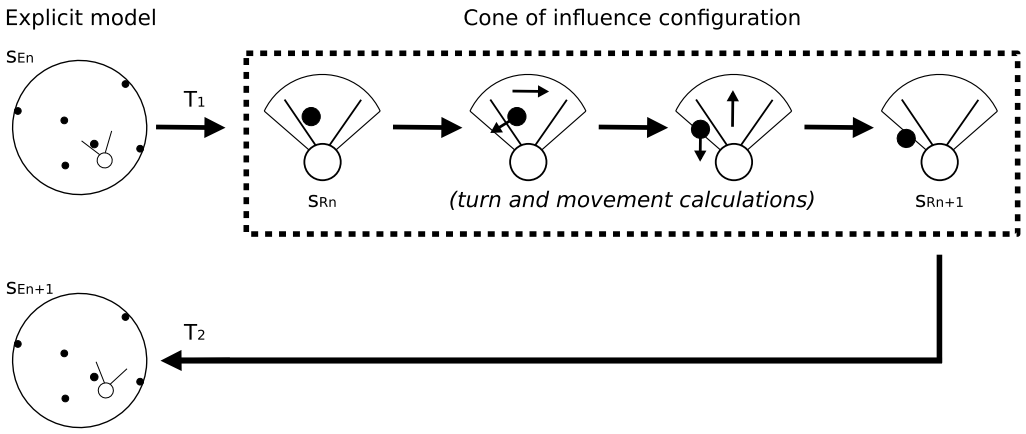


Figure 5.11: Visualisation of transition function $\mathcal{F}_E$.

The transition $\mathcal{F}_E$ is further illustrated in Figure 5.11. On the left, there is an example of a state transition in the Explicit model. The transition is from state s_{En} to s_{En+1}. Firstly, state s_{En} is translated to s_{Rn}. From here, in the *Cone of influence representation*, the transition function $\mathcal{F}_R$ is illustrated. Once the *turn and movement calculations* are preformed, the resulting state s_{Rn+1} is reflected

back to the Explicit model using the translation function $\mathcal{T}_2$. In $\mathcal{T}_2$, the state s_{Rn+1} is combined with the key to arrive at the successor state of s_{En}, s_{En+1}.

5.4.2 Translation function $\mathcal{T}_1$

The translation function $\mathcal{T}_1$ maps a state from the Explicit model (s_{En}) to an equivalent state in the Relative model (s_{Rn}). To calculate this translation, we take the coordinate of the agent in the Explicit model and isolate a COI around it. Once in this representation the translation is complete. The calculations for this are as follows.

Coordinates of the nearest obstacle

In the first part of the translation we determine the closest obstacle to the agent, and whether that obstacle is in the agent's COI. The following definitions describe an obstacle that is within an agent's COI.

- Let $\mathcal{O}$ represent the coordinates of all the obstacles in the Explicit model such that $\mathcal{O} \subseteq \mathcal{D} \times \mathcal{A}$.
- Let $\mathcal{Z}$ represent all the coordinates that make up the agent's COI in the Explicit model such that $\mathcal{Z} \subseteq \mathcal{D} \times \mathcal{A}$.
- Let $ob = \mathcal{O} \cap \mathcal{Z}$ where $|ob| \in \{\emptyset, 1\}$. The value of $|ob|$ determines whether there is an obstacle in the COI.

If there is an obstacle in the COI then the polar coordinates of its centre point are known, we denote them as $(oA,\ oD)$. Using these coordinates and the coordinates $(eA,\ eD)$ and angle rA from state s_{En} we can begin converting to the equivalent state in the Relative model, s_{Rn}. Note that when $|ob| = \emptyset$, the state is mapped to $relA = 0$ and $relD = 0$, implying that the agent is in *free-space*.

Figure 5.12 shows the triangles which represent the Euclidean distances of the agent and the obstacle relative to the centre of the environment. The coordinates of the agent and the obstacle are used to generate them. Triangles $\triangle_1$ and $\triangle_2$

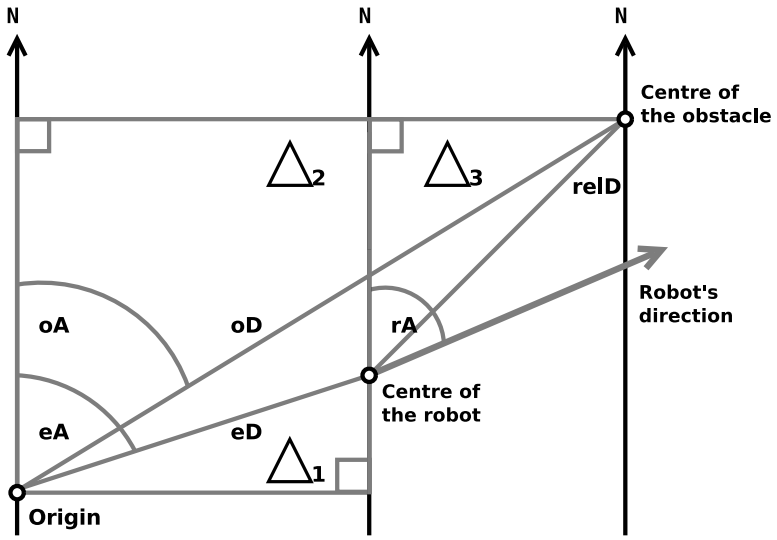


Figure 5.12: Triangles representing the calculations to convert from the Explicit to the Relative model.

are made between the centre of the environment and the position of the agent or obstacle, respectively. Triangle $\triangle_3$ is calculated from triangles $\triangle_1$ and $\triangle_2$. It is used to calculate the agent's COI, and the position of the obstacle inside it –relative to the centre of the agent.

Euclidean distances of the agent

If $|ob| = 1$, then we calculate the Euclidean distances from the axes to the centre of the agent. There are two Euclidean axes. One runs in line with polar axis (due North), the other runs perpendicular to this, passing horizontally through the pole. The measurements from these axes to the centre of the agent (in triangle $\triangle_1$) are represented by line lengths $lOrg$ and $hOrg$, length horizontally and height vertically respectively. These lines are combined with the line eD, representing the distance from the origin to the centre of the agent, to form a triangle. The angle formed at the origin is oZ. Calculations for $\triangle_1$ are shown in Equation 5.6.

$$
\begin{aligned}
oZ &= eA\%90 \\
(lOrg, hOrg) &= \begin{cases}
(eD, 0), & \textit{if } eA \in \{90, 270\} \\
(0, eD), & \textit{if } eA \in \{0, 180\} \\
((\sin(oZ) * eD), & \\
(\cos(oZ) * eD)), & \textit{if } eA \in \{a : (0 \leq a \leq 90) \\
 & \cup (180 \leq a \leq 270)\} \\
((\cos(oZ) * eD), & \\
(\sin(oZ) * eD)), & otherwise
\end{cases}
\end{aligned}
\tag{5.6}
$$

Euclidean distances of the obstacle

Next we calculate the Euclidean distances from the axes to the centre of the obstacle that is nearest to the agent (where the obstacle's coordinate is represented by ob). The measurement from these axes to the centre of ob (in triangle $\triangle_2$) are represented by line lengths $lNew$ and $hNew$, length horizontally and height vertically respectively. These lines are combined with the line oD to represent the distance from the origin to the centre of the nearest obstacle, and they form the triangle $\triangle_2$. The angle formed at the origin is nZ. The values of triangle $\triangle_2$ are calculated as shown in Equation 5.7.

$$
\begin{aligned}
nZ &= oA\%90 \\
(lNew, hNew) &= \begin{cases}
(oD, 0), & if\, oA \in \{90, 270\} \\
(0, oD), & if\, oA \in \{0, 180\} \\
((\sin(nZ) * oD), & \\
(\cos(nZ) * oD)), & if\, oA \in \{a : (0 \leq a \leq 90) \\
 & \cup (180 \leq a \leq 270)\} \\
((\cos(nZ) * oD), & \\
(\sin(nZ) * oD)), & otherwise
\end{cases}
\end{aligned}
\tag{5.7}
$$

Distances between the agent and obstacle

Using triangles $\triangle_1$ and $\triangle_2$, we calculate the distance between the obstacle and the agent to generate triangle $\triangle_3$. This is done by calculating the horizontal and vertical distances between them, $lFin$ and $hFin$ respectively. Angle fZ is formed at the centre of the robot of the new triangle $\triangle_3$, which has sides $lFin$ and $hFin$. The calculations for triangle $\triangle_3$ are shown in Equation 5.8.

$$
\begin{aligned}
lFin \quad = \quad & \begin{cases} lOrg + lNew, & if\ ((eA \in \{a : (0 \leq a < 180)\} \cap \\ & & oA \in \{b : (180 < b < 360)\}) \cup \\ & & (eA \in \{a : (180 < a < 360)\} \cap \\ & & oA \in \{b : (0 \leq b < 180)\})) \\ |lOrg - lNew|, & otherwise \end{cases} \\
hFin \quad = \quad & \begin{cases} hOrg + hNew, & if\ (((eA \in \{a : (0 \leq a < 90) \\ & & \cup (270 < a < 360)\}) \\ & & \cap (oA \in \{b : (90 < b < 270)\})) \cup \\ & & (eA \in \{a : (90 < a < 270)\} \cap \\ & & (oA \in \{b : (0 \leq b < 90) \cup \\ & & (270 < b < 360)\}))) \\ |hOrg - hNew|, & otherwise \end{cases} \\
fZ \quad = \quad & \arctan(hFin/lFin)
\end{aligned}
\tag{5.8}
$$

Relative position of the agent to the obstacle

Next we calculate the relative position of the agent to the obstacle, horizontally (fR) and vertically (fU). Calculating whether the agent is further up and whether it is further right of the obstacle is necessary for calculating the angle of the obstacle relative to the agent ($relA$). The values of fU and fR are calculated as shown in Equation 5.9.

$$
\begin{aligned}
fR \quad = \quad & \begin{cases}
1, & \textit{if } ((eA \in \{a : (0 \leq a \leq 180)\} \cap \\
& oA \in \{b : (0 \leq b \leq 180)\} \cap (lOrg > lNew)) \cup \\
& (eA \in \{a : (180 \leq a < 360)\} \cap \\
& oA \in \{b : (180 \leq b < 360)\} \cap (lOrg < lNew)) \cup \\
& (eA \in \{a : (0 \leq a < 180)\} \cap oA \in \{b : (180 < b < 360)\})) \\
0, & \textit{otherwise}
\end{cases} \\
fU \quad = \quad & \begin{cases}
1, & \textit{if } (((eA \in \{a : (0 \leq a \leq 90) \cup (270 \leq a < 360)\}) \\
& \cap (oA \in \{b : (0 \leq b \leq 90) \cup (270 \leq b < 360)\}) \\
& \cap (hOrg > hNew)) \cup \\
& (eA \in \{a : (90 \leq a \leq 270)\} \cap \\
& oA \in \{b : (90 \leq b \leq 270)\} \\
& \cap (hOrg < hNew)) \cup \\
& ((eA \in \{a : (0 \leq a \leq 90) \cup (270 \leq a < 360)\}) \cap \\
& oA \in \{b : (90 \leq b \leq 270)\})) \\
0, & \textit{otherwise}
\end{cases}
\end{aligned}
\tag{5.9}
$$

Polar angle from the agent to the obstacle

Once we have the position of the agent relative to the obstacle and $\triangle_1$ and $\triangle_2$, we can calculate the relative angle and the relative distance from the agent to the obstacle ($relA$ and $relD$). The angle $relA$ is measured from the farthest anti-clockwise point in the agent's COI (40° anticlockwise from the direction that the agent is facing). The distance $relD$ is the distance from the centre of the agent to the centre of the obstacle. The calculations for $relA$ and $relD$ are shown in Equation 5.10.

$$\begin{aligned} relA &= ((90 + (180 * fR) + (2 * fZ * |fR - fU|) - fZ) - rA + 400)\%360 \\ relD &= \sqrt{(hFin^2 + lFin^2)} \end{aligned} \tag{5.10}$$

5.4.3 Transition function $\mathcal{F}_R$

The function $\mathcal{F}_R$ represents a transition in the Relative model; mapping from a state s_{Rn} to s_{Rn+1} where $(s_{Rn}, s_{Rn+1}) \in R_R$. This function represents the agent's reaction to its antennas' signals, and its forward movement.

The first part of the function involves calculating the signal difference between the antennas, $aDif'$ (the previous difference being $aDif$). Using this and the agent's learning weight lW, next we calculate the orientation of the agent relative to the obstacle, producing $relA^*$. The agent is then turned using $relA^*$. Finally, the agent is moved forward, which produces the values of $relA'$ and $relD'$.

Note that $relA^*$ is not the final relative angle, because when the agent is moved forward the relative angle is changed; i.e., the final relative angle is $relA'$.

Difference in antenna signals

Calculating $aDif'$ uses the coordinates ($relD$, $relA$). If $relA$ and $relD$ are both 0, then $aDif'$ is assigned 0, otherwise we use the Equation 5.11.

Each section of an antenna produces a different signal, so it is necessary to calculate the specific section that the obstacle is contacting. For each section of antenna we identify a set of coordinates, where a set consists of the centre-points of all obstacles that could touch that section of antenna (see $\mathcal{S}_L(x)$ and $\mathcal{S}_R(x)$ in Section 5.3.1). It is important to note that the closer the section of antenna is to the centre of the agent, the higher the precedence its signal has when calculating $aDif'$; e.g., if an obstacle covers coordinates in two sections of an antenna, the resulting signal is the value from the section that is closer to the centre of the agent. The calculation of $aDif'$ is as follows.

$$
aDif' = \begin{cases}
-6, & if((relA, relD) \in \mathcal{S}_{L1}) \\
-5, & if(((relA, relD) \in \mathcal{S}_{L2}) \cap ((relA, relD) \backslash \mathcal{S}_{L1})) \\
-4, & if(((relA, relD) \in \mathcal{S}_{L3}) \cap ((relA, relD) \backslash \mathcal{S}_{L2}) \cap \\
 & ((relA, relD) \backslash \mathcal{S}_{L1})) \\
-3, & if(((relA, relD) \in \mathcal{S}_{L4}) \cap ((relA, relD) \backslash \mathcal{S}_{L3}) \cap \\
 & ((relA, relD) \backslash \mathcal{S}_{L2})) \\
-2, & if(((relA, relD) \in \mathcal{S}_{L5}) \cap ((relA, relD) \backslash \mathcal{S}_{L4}) \cap \\
 & ((relA, relD) \backslash \mathcal{S}_{L3})) \\
-1, & if(((relA, relD) \in \mathcal{S}_{L6}) \cap ((relA, relD) \backslash \mathcal{S}_{L5}) \cap \\
 & ((relA, relD) \backslash \mathcal{S}_{L4})) \\
6, & if((relA, relD) \in \mathcal{S}_{R1}) \\
5, & if(((relA, relD) \in \mathcal{S}_{R2}) \cap ((relA, relD) \backslash \mathcal{S}_{R1})) \\
4, & if(((relA, relD) \in \mathcal{S}_{R3}) \cap ((relA, relD) \backslash \mathcal{S}_{R2}) \cap \\
 & ((relA, relD) \backslash \mathcal{S}_{R1})) \\
3, & if(((relA, relD) \in \mathcal{S}_{R4}) \cap ((relA, relD) \backslash \mathcal{S}_{R3}) \cap \\
 & ((relA, relD) \backslash \mathcal{S}_{R2})) \\
2, & if(((relA, relD) \in \mathcal{S}_{R5}) \cap ((relA, relD) \backslash \mathcal{S}_{R4}) \cap \\
 & ((relA, relD) \backslash \mathcal{S}_{R3})) \\
1, & if(((relA, relD) \in \mathcal{S}_{R6}) \cap ((relA, relD) \backslash \mathcal{S}_{R5}) \cap \\
 & ((relA, relD) \backslash \mathcal{S}_{R4})) \\
0, & otherwise
\end{cases} \tag{5.11}
$$

Angle turned

Next we use $aDif'$ to calculate how the agent's turn affects the angle of the obstacle relative to the agent, $relA^*$.

$$relA^* = \begin{cases} relA + (aDif' * lW), & if(0 \leq (relA + (aDif' * lW)) \leq 80) \\ 0, & otherwise \end{cases} \quad (5.12)$$

Move forward

Now that the agent's turn response has been calculated we can move the agent forward to calculate the new relative coordinate ($relA', relD'$), and the new learning weight (lW').

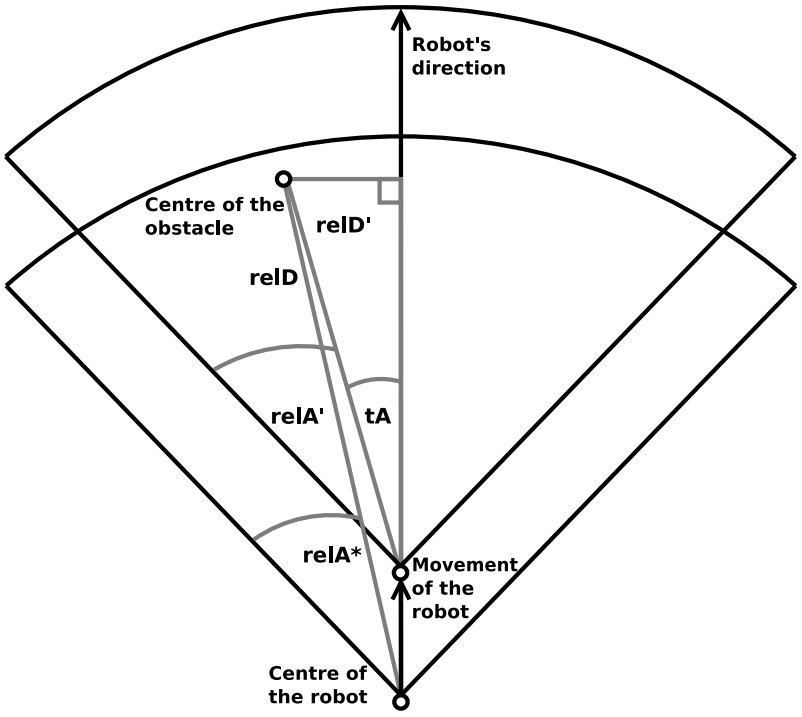


Figure 5.13: Triangles representing the calculations for a transition in the Relative model.

Figure 5.13 illustrates how $relA'$ and $relD'$ are calculated as the agent moves forward. Moving forward forms a new, smaller triangle with the vertical line projected from the pole. This is used to calculate the new relative angle and distance to the obstacle, $relA'$ and $relD'$ respectively. The learning weight lW' is calcu-

lated from the signal difference and the relative distance to the obstacle, Dif' and $relD'$ respectively.

$$relD' = \sqrt{(relD^2 - 2(relD * cos(|relA^* - 40|)) + 1)} \tag{5.13}$$

$$tA = \arctan(\frac{\sqrt{relD^2 - (relD * \cos(|relA^* - 40|))^2}}{(relD * cos(|relA^* - 40|) - 1)}) \tag{5.14}$$

$$relA' = \begin{cases} 40 - tA, & if\ (tA < 40) \\ 40 + tA, & otherwise \end{cases} \tag{5.15}$$

$$lW' = \begin{cases} lW + 1, & if\ (((aDif' \geq 6) \cup (aDif' \leq -6)) \cap \\ & ((aDif \leq 5) \cap (aDif \geq -5) \cap (aDif \neq 0))) \\ lW, & otherwise \end{cases} \tag{5.16}$$

5.4.4 Translation function $\mathcal{T}_2$

The translation function $\mathcal{T}_2$ is applied to a state in the Relative model to map it to a state in the Explicit model. It uses the values $aDif$ and lW from the state s_{Rn}, and eA, eD, and rA from the key, $\mathcal{K}_n$ (see Section 5.4.1). Because every state in the Relative model can be mapped to a number of possible states in the Explicit model, $\mathcal{K}_n$ is used to identify the exact state to map to, as it references the previous state in the Explicit model. Translation function $\mathcal{T}_2$ uses the state s_{Rn+1} and $\mathcal{K}_n$ to calculate s_{En+1}, where state $s_{En+1} = (eA', eD', rA', aDif', lW')$.

Figure 5.14 represents the variables used in this translation. Three right angle triangles are used to represent the angles and distances between the *origin*, the original position of the agent, and the new position of the agent. They are the triangles comprised of: $\triangle_1$, the North line at original position of the agent and the origin; $\triangle_2$, the new position of the agent and North line at its original position; and $\triangle_3$, the new position of the agent and the North line at the origin. State s_{En+1} is calculated with reference to these triangles, as follows.

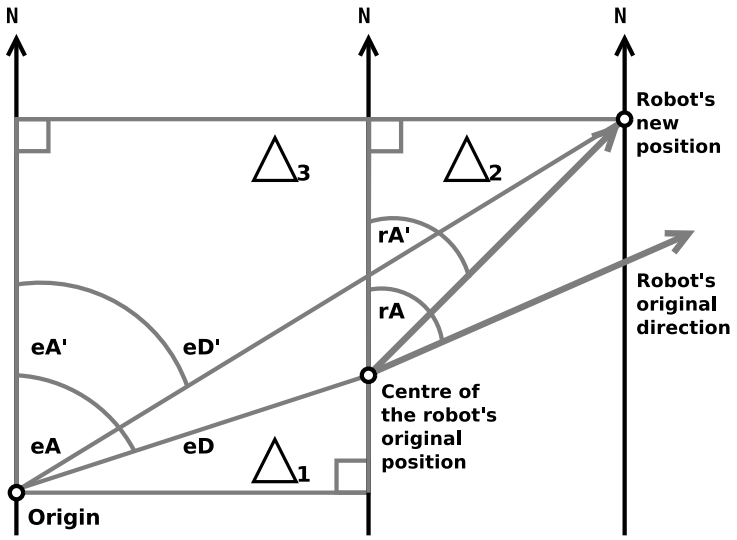


Figure 5.14: Triangles representing the calculations for the translation from the Relative to the Explicit model.

Orientation of agent

Variables $aDif'$ and lW (from s_{Rn}), and angle rA (from $\mathcal{K}_n$) are used to calculate the new angle that the agent is facing rA', as indicated in Equation 5.17.

$$rA' \quad = \quad (rA + (aDif' * lW) + 360)\%360 \tag{5.17}$$

Euclidean distances of the original position of the agent

Next, we use the angle and distance from the agent's original position to the centre of the environment, eA and eD respectively, to generate $\triangle_1$.[9] First, we calculate the Euclidean distances from the axes to the centre of the agent. The measurements from these axes to the centre of the agent are represented by line lengths $lOrg$ and $hOrg$, length horizontally and height vertically respectively. These lines are combined with the line eD, which represents the distance from the origin to the centre of the agent, to form the triangle $\triangle_1$. The angle formed at the origin is oZ. The values are calculated as in Equations 5.18.

[9]Note that this was already calculated in $\mathcal{T}_1$ (in Section 5.4.2). Hence, in the model we simply store and then reuse this, opposed to recalculating it.

$$
\begin{aligned}
oZ &= eA\%90 \\
(lOrg, hOrg) &= \begin{cases}
(eD, 0), & if\, eA \in \{90, 270\} \\
(0, eD), & if\, eA \in \{0, 180\} \\
((\sin(oZ) * eD), & \\
(\cos(oZ) * eD)), & if\, eA \in \{a : (0 \leq a \leq 90) \\
 & \cup (180 \leq a \leq 270)\} \\
((\cos(oZ) * eD), & \\
(\sin(oZ) * eD)), & otherwise
\end{cases}
\end{aligned} \tag{5.18}
$$

Euclidean distances of the agent's new position relative to its original

We now calculate the Euclidean distances from the new axes centred at the agent's original position. Lines are drawn perpendicular from each axis to the centre of the agent (horizontally and vertically). The measurements of these lines are represented by the variables $lNew$ and $hNew$, length horizontally and height vertically respectively. These lines are combined with a line representing the agent's movement in one time-step (1 unit at an angle of rA' from North) to form the triangle $\triangle_2$. The angle it forms at the new axis is nZ. The values are calculated in Equations 5.19.

$$
\begin{aligned}
nZ &= rA' \% 90 \\
(lNew, hNew) &= \begin{cases}
(1,0): & if\, rA' \in \{90, 270\} \\
(0,1): & if\, rA' \in \{0, 180\} \\
((\sin(nZ) * 1), & \\
(\cos(nZ) * 1)): & if\, rA' \in \{a : (0 \leq a \leq 90) \\
 & \cup (180 \leq a \leq 270)\} \\
((\cos(nZ) * 1), & \\
(\sin(nZ) * 1)): & otherwise
\end{cases}
\end{aligned}
\tag{5.19}
$$

Euclidean distances of the new position of the agent relative to the axes

After calculating triangles $\triangle_1$ and $\triangle_2$, we use their measurements to calculate the distance between the new position of the agent and the environment's axes to make triangle $\triangle_3$. We do this in the same way as with the original position, by calculating the horizontal and vertical distances with $lFin$ and $hFin$, respectively. Here, $lFin$ and $hFin$ form the sides of triangle $\triangle_3$, with fZ being the angle made at the origin. These calculations are shown in Equations 5.20.

$$
\begin{aligned}
lFin \quad = \quad & \begin{cases} |lOrg - lNew|, & if((eA \in \{a : (0 \leq a < 180)\} \cap \\ & & rA' \in \{b : (180 < b < 360)\}) \cup \\ & & (eA \in \{a : (180 < a < 360)\} \cap \\ & & rA' \in \{b : (0 \leq b < 180)\})) \\ lOrg + lNew, & otherwise \end{cases} \\
hFin \quad = \quad & \begin{cases} |hOrg - hNew|, & if(((eA \in \{a : (0 \leq a < 90) \\ & & \cup (270 < a < 360)\}) \\ & & \cap (rA' \in \{b : (90 < b < 270)\})) \cup \\ & & ((eA \in \{a : (90 < a < 270)\}) \cap \\ & & (rA' \in \{b : (0 \leq b < 90) \cup \\ & & (270 < b < 360)\}))) \\ hOrg + hNew, & otherwise \end{cases} \\
fZ \quad = \quad & \arctan(hFin/lFin)
\end{aligned}
\tag{5.20}
$$

Calculate eD'

The hypotenuse of $\triangle_3$ is the distance from the centre of the environment to the centre of the agent eD', and is calculated by Equation 5.21.

$$
eD' \quad = \quad \sqrt{(hFin^2 + lFin^2)} \tag{5.21}
$$

Quadrant of the agent after moving

Next we calculate the new position of the agent relative to its original position, farther right horizontally (fR) and farther up vertically (fU). The calculations shown in Equations 5.22.

$$
\begin{aligned}
fR \quad = \quad & \begin{cases}
0, & if(((eA \in \{a : (0 \leq a < 90)\}) \cap (rA' \in \{b : (180 \leq a \leq 360)\}) \\
& \cap (lNew > lOrg)) \cup \\
& ((eA \in \{a : (90 \leq a < 180)\}) \cap (rA' \in \{b : (180 \leq a \leq 360)\}) \\
& \cap (lNew > lOrg)) \cup \\
& ((eA \in \{a : (180 \leq a < 270)\}) \cap ((rA' \in \{b : (180 \leq a \leq 360)\}) \\
& \cup (lNew < lOrg))) \cup \\
& ((eA \in \{a : (270 \leq a < 360)\}) \cap ((rA' \in \{b : (180 \leq a \leq 360)\}) \\
& \cup (lNew < lOrg)))) \\
1, & otherwise
\end{cases} \\
fU \quad = \quad & \begin{cases}
0, & if(((eA \in \{a : (0 \leq a < 90)\}) \cap (rA' \in \{b : (90 \leq a \leq 270)\}) \\
& \cap (hNew > hOrg)) \cup \\
& ((eA \in \{a : (90 \leq a < 180)\}) \cap ((rA' \in \{b : (90 \leq a \leq 270)\}) \\
& \cup (hNew < hOrg))) \cup \\
& ((eA \in \{a : (180 \leq a < 270)\}) \cap ((rA' \in \{b : (90 \leq a \leq 270)\}) \\
& \cup (hNew < hOrg))) \cup \\
& ((eA \in \{a : (270 \leq a < 360)\}) \cap (rA' \in \{b : (90 \leq a \leq 270)\}) \\
& \cap (hNew > hOrg))) \\
1, & otherwise
\end{cases}
\end{aligned}
\tag{5.22}
$$

Calculate eA'

Calculating the polar angle of the agent relative to the environment (eA') depends on where the new position of the agent is relative to its original position. Hence, we use fU and fR to calculate it in Equations 5.23.

$$
eA' \quad = \quad \begin{cases}
0, & if(((eA == 0) \cap (rA' == 180) \\
 & \cap(hNew \leq hOrg)) \cup \\
 & ((eA == 180) \cap (rA' == 0) \\
 & \cap(hNew \geq hOrg)) \cup \\
 & (((eA == 90) \cap (rA' == 270)) \\
 & \cup((eA == 270) \cap (rA' == 90)) \\
 & \cap(lNew == lOrg))) \\
90, & if(((eA == 270) \cap (rA' == 90) \\
 & \cap(lNew > lOrg)) \cup \\
 & ((eA == 90) \cap (rA' == 270) \\
 & \cap(lNew < lOrg))) \\
180, & if(((eA == 0) \cap (rA' == 180) \\
 & \cap(hNew > hOrg)) \cup \\
 & ((eA == 180) \cap (rA' == 0) \\
 & \cap(hNew < hOrg))) \\
270, & if(((eA == 90) \cap (rA' == 270) \\
 & \cap(lNew > lOrg)) \cup \\
 & ((eA == 270) \cap (rA' == 90) \\
 & \cap(lNew < lOrg))) \\
(90 - fZ), & if((fR == 1) \cap (fU == 1)) \\
(90 + fZ), & if((fR == 1) \cap (fU == 0)) \\
(270 - fZ), & if((fR == 0) \cap (fU == 0)) \\
(270 + fZ), & if((fR == 0) \cap (fU == 1)) \\
rA', & otherwise
\end{cases} \tag{5.23}
$$

Once eA', eD' and rA' are calculated, we combine them with $aDif'$ and lW'

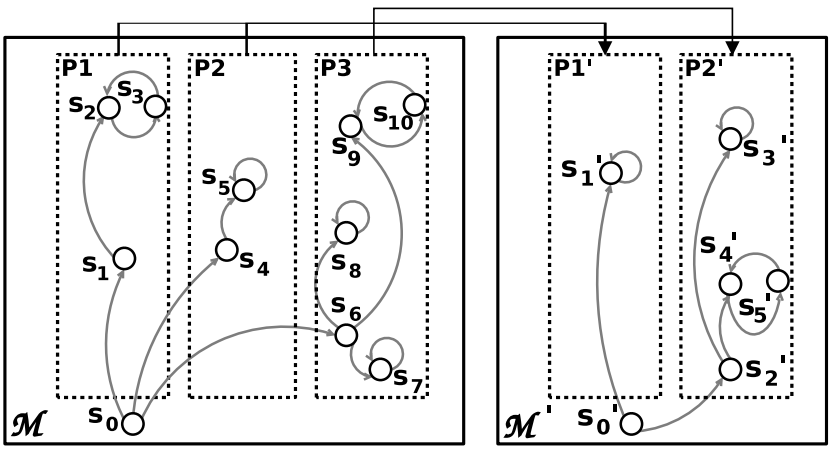


Figure 5.15: Simulation relation.

to produce the successor state in the Explicit model, s_{En+1}.

5.5 Simulation relation

A simulation relation [78] is a relation between the states of two Kripke structures, $\mathcal{M}$ and $\mathcal{M}'$ that preserves properties expressed by LTL formulas. If we have a model $\mathcal{M}$ that is too large for us to verify properties for, and it is possible to create a model $\mathcal{M}'$ that simulates $\mathcal{M}$, then we can verify properties of $\mathcal{M}'$ and thus infer that they hold for $\mathcal{M}$. We illustrate the concept of a simulation relation (H) via Figure 5.15. The simulation relation here is:

$$\begin{aligned} H \quad = \quad & \{(s_0, s'_0), (s_1, s'_1), (s_2, s'_1), (s_3, s'_1), (s_4, s'_1), (s_5, s'_1), \\ & (s_6, s'_2), (s_7, s'_3), (s_8, s'_3), (s_9, s'_4), (s_{10}, s'_5)\} \end{aligned} \tag{5.24}$$

Note that paths $P1$ and $P2$ in $\mathcal{M}$ are mapped to paths $P1'$ in $\mathcal{M}'$, and $P3$ to $P2'$.

In this section we introduce a slightly modified form of simulation, namely ϕ-*simulation*. Here ϕ represents a set of atomic propositions: AP_ϕ. We show that ϕ-simulation applies between the Kripke structures of the Explicit and Relative models. The set AP_ϕ contains all atomic propositions from the variables that the

models have in common. The common variables between state vectors are $aDif$ (antennas' signal difference) and lW (learning weight). Given a property ψ that is represented by a formula composed of atomic propositions AP_ψ, and where these are only atomic propositions relating to $aDif$ and lW, such that $AP_\psi \subseteq AP_\phi$. Then, if the Relative model ϕ-simulates the Explicit model, ψ holding for the Relative model implies that it also holds for the Explicit model.

Herein, we prove the existence of this ϕ-simulation relation between the Explicit and Relative models. We begin this proof by asserting that the translation function $\mathcal{T}_1$ maps every state in the Explicit model ($\mathcal{M}_E$) to a state in the Relative model ($\mathcal{M}_R$). Furthermore, we propose that the ϕ-simulation relation applies to the models as $\mathcal{M}_R$ ϕ-simulates $\mathcal{M}_E$ (denoted by $\mathcal{M}_E \preceq_\phi \mathcal{M}_R$). We use H_ϕ to denote the ϕ-simulation relation where $H_\phi \subseteq S_E \times S_R$ and $H_\phi = \{(s_{En}, \mathcal{T}_1(s_{En})) \mid s_{En} \in S_E\}$.

5.5.1 ϕ-Simulation relation

Our relation ϕ-simulation relation, is defined as follows.

Let AP_ϕ denote a set of all atomic propositions that relate to the common variables $aDif$ and lW. For any state s, let $L_\phi(s)$ denote the label of s with respect to AP_ϕ (i.e., the set of propositions from AP_ϕ that are true at s). The following definition is adapted from [7].

Definition 5.3. Given two structures $\mathcal{M}= (S, s_0, R, L)$ and $\mathcal{M}'= (S', s'_0, R', L')$ whose sets of propositions contain AP_ϕ, a relation $H_\phi \subseteq (S \times S')$ is a ϕ-simulation relation between $\mathcal{M}$ and $\mathcal{M}'$ if and only if for all $H_\phi(s, s')$:

1. $L_\phi(s) = L_\phi(s')$
2. For every state $s_1 \in S$ such that $R(s, s_1)$, there is a state $s'_1 \in S'$ with the property that $R'(s', s'_1)$ and $H_\phi(s_1, s'_1)$.

We say that $\mathcal{M} \preceq_\phi \mathcal{M}'$ if there exists a ϕ-simulation relation H_ϕ such that $H_\phi(s_0, s'_0)$.

From this we derive the following theorem (also adapted from [7]).

Theorem 5.4. Suppose $\mathcal{M} \preceq_\phi \mathcal{M}'$. Then for every LTL formula that represents a property ψ, where $AP_\psi \subseteq AP_\phi$: $\mathcal{M}' \models \psi$ implies $\mathcal{M} \models \psi$.

5.5.2 Proof that our abstraction is sound

In order to prove that satisfaction of an LTL formula for the Relative model implies its satisfaction for any Explicit model with an environment of same type, we must demonstrate that there is a ϕ-simulation relation, H_ϕ, between our models such that $\mathcal{M}_E \preceq_\phi \mathcal{M}_R$. Therefore, we need to determine that the relation H_ϕ (i.e. the set of pairs of states (s_{En}, s_{Rn})) satisfies the conditions of Definition 5.3. In this section we use the term model to denote the underlying Kripke structures associated with a PROMELA specification (see Definitions 5.1 and 5.2 for the Kripke structures of the Explicit and Relative models respectively).

Before we define this ϕ-simulation, we justify our approach with reference to Figure 5.16. Consider transition (s_{En}, s_{En+1}) in the left model ($\mathcal{M}_E$). State s_{En} is mapped to a state s_{Rn} in the right model ($\mathcal{M}_R$) using function $\mathcal{T}_1$, and then there is a transition from s_{Rn} to s_{Rn+1} via $\mathcal{F}_R$. The function $\mathcal{F}_R$ takes the position of an obstacle relative to the centre of the agent and calculates the agent's turn, movement, and learning to produce the next state. The relative position of the obstacle is used to calculate the new antenna difference signal $aDif'$, which in reference to the code is denoted as `x` ($aDif$ and lW in reference to the code are denoted as `Prev_x` and ω_d respectively). Once s_{Rn+1} is calculated, it is mapped back to s_{En+1} via $\mathcal{T}_2$ using the key K_n. The key is used to identify a unique successor state in $\mathcal{M}_E$, as a state in $\mathcal{M}_R$ can be mapped to many states. Identified by K_n are the original coordinates of the agent relative to the centre of the environment. Given this information and the result of the agent's turn, movement, and learning from $\mathcal{F}_R$, we can now calculate s_{En+1} (the unique successor state of s_{En}).

Note that s_{En} to s_{Rn}, and s_{En+1} to s_{Rn+1} can be matched because their COI representations hold the same information. Specifically, the relative position between an agent and an obstacle is the same in both explicit and relative representations. Therefore, the antenna signals (`x` and `Prev_x`) and the learning weight (ω_d) are also the same –as they are only affected by the relative position of an obstacle. (These functions are defined in Section 5.4.)

To formally prove the existence of a ϕ-simulation relation we must fulfil the conditions of Definition 5.3. For our proof of a ϕ-simulation we determine that

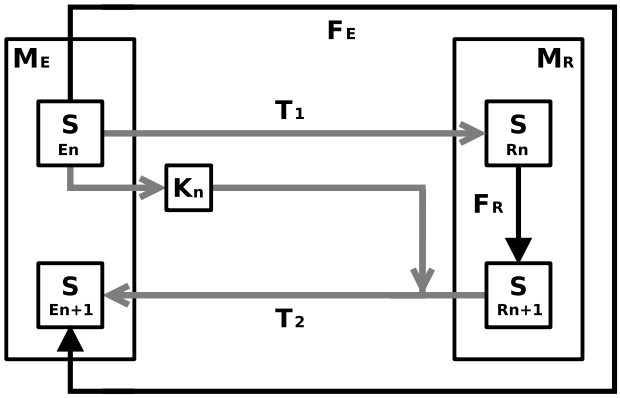


Figure 5.16: Deterministic function mapping. Grey arrows indicate translations between models, and black arrows indicate transitions within models.

the set AP_ϕ contains all atomic propositions from the variables $aDif$ and lW. Specifically relating to the code, these variables are `x`, `Prev_x`, and ω_d. Both properties `1P` and `2P`, that we check in $\mathcal{M}_E$ and $\mathcal{M}_R$ (see Sections 4.3 and 6.1), refer to these variables and, hence, are contained within the set AP_ϕ.

We define $H_\phi \subset (S_E \times S_R)$ as follows: $H_\phi = \{(s_{En}, s_{Rn}) : s_{En} \in S_E \text{ and } s_R = \mathcal{T}_1(s_E)\}$. For any state s_E, the agent's learning weight and the relative position of any obstacle to it are identical in s_E and $\mathcal{T}_1(s_E)$; so `x`, `Prev_x`, and ω_d are the same for both of states. Hence the atomic propositions in the formulas `1P` and `2P` are the same in s_E and $\mathcal{T}_1(s_E)$. So for any $(s_E, s_R) \in H_\phi$, $L_\phi(s_E) = L_\phi(s_R)$. (Fulfilling Definition 5.3.1.)

Now consider Definition 5.3.2, where for a transition (s_{En}, s_{En+1}) in $\mathcal{M}_E$ we must show that, if $H_\phi(s_{En}, s_{Rn})$ then there exists an s_{Rn+1} such that (s_{Rn}, s_{Rn+1}) is a transition in $\mathcal{M}_R$ and $H_\phi(s_{En+1}, s_{Rn+1})$. We consider separately the cases where, at s_{En} there is/is not an obstacle in the COI representation.

If there is an obstacle in the COI, the transition from s_{Rn} in $\mathcal{M}_R$ is purely deterministic. Function $\mathcal{T}_2$ is the inverse of $\mathcal{T}_1$ (it maps a state from the $\mathcal{M}_R$ to one in the $\mathcal{M}_E$, using K_n as defined in Figure 5.16). Since $\mathcal{T}_1(s_{En+1}) = s_{Rn+1}$, then $H_\phi(s_{En+1}, s_{Rn+1})$.

Suppose then that there is no obstacle in the COI. Then s_{En} is mapped via $\mathcal{T}_1$ to a *free-space* state in $\mathcal{M}_R$. That is, state s_{Rn} in Figure 5.17 such that $H_\phi(s_{En}, s_{Rn})$.

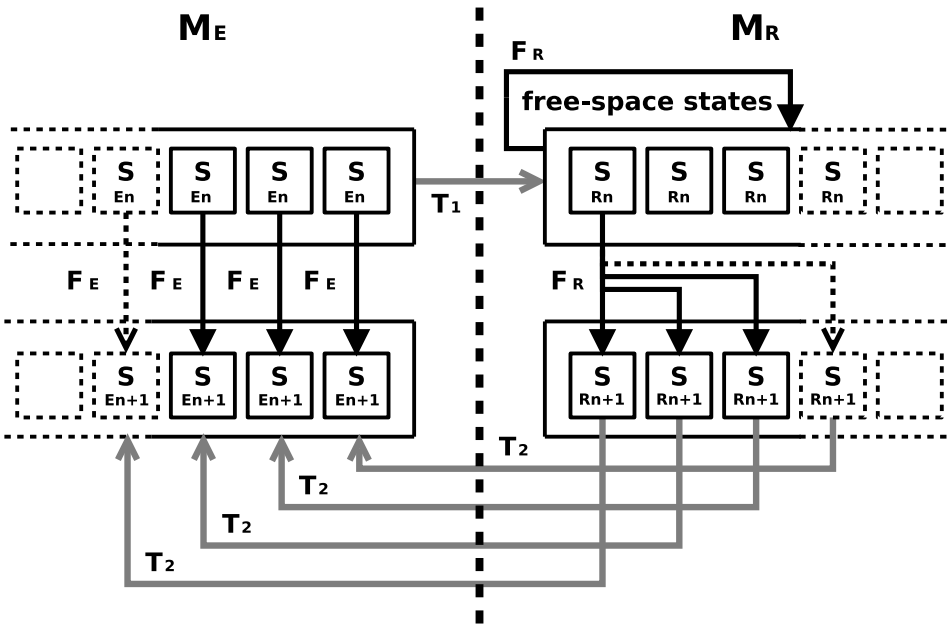


Figure 5.17: Nondeterministic function mapping. Grey arrows indicate translations between models, and black arrows indicate transitions within models.

The set of free-space states in $\mathcal{M}_R$ have no obstacles in their COIs and the only difference between them is the value of the learning weight ω_d. From a free-space state s_{Rn} there is a nondeterministic choice to transition back to itself (no obstacle in the COI and the same ω_d value), or to any state in which there is an obstacle at the boundary of the COI (see the right hand side of Figure 5.17, $\mathcal{M}_R$).

Note that all states s_{En} in $\mathcal{M}_E$ (for which there is no obstacle in the COI) are mapped via $\mathcal{T}_1$ to a free-space state s_{Rn} in $\mathcal{M}_R$. The nondeterministic choice of transitions from s_{Rn} are illustrated by the transition from free-space to free-space (s_{Rn} to s_{Rn}), and transitions to the different s_{Rn+1} states.

For the possible s_{Rn+1} states, all legal positions of an obstacle at the boundary of the COI are considered (see Section 5.2.2). Hence, the available set of transitions from a free-space state encompasses all possible, legal and subsequent encounters with an obstacle. Therefore, for every transition (s_{En}, s_{En+1}) in $\mathcal{M}_E$, where $H_\phi(s_{En}, s_{Rn})$, there is a matching transition F_R in $\mathcal{M}_R$ such that $F_R(s_{Rn}) = s_{Rn+1}$ and $\mathcal{T}_2(s_{Rn+1}) = s_{En+1}$, and hence $H_\phi(s_{En+1}, s_{Rn+1})$. (Fulfilling Definition 5.3.2.)

Chapter 6

Application of Agent-centric abstraction for PROMELA

In this chapter we describe the implementation of our Agent-centric abstraction for a PROMELA model.

6.1 PROMELA Relative model

The Relative model is derived directly from the specification of the ABL system described in Section 2.2.3, and generating it involves the rigorous calculations dictated by the Agent-centric abstraction. We model the same system as for the Explicit model in order for a direct comparison of approaches; where the verifications involve the same properties, and the results are presented in the same format.

We give outline code for the Relative model specification in Figure 6.1. The `C` macro functions are included in the Appendix C.3.

The process `moving` represents the agent in the COI, where the `do` loop is used to determine between situations when the agent encounters an obstacle and when they are in free-space.

If there is an obstacle in the COI then the `C` macros for responding, moving, and learning are called (these are similar functions to those described in the Ex-

```
/*Abstract  Model: Using Functions (Macros)*/

c_decl { #include <math.h> }
#include "absInlines.txt"

int obDist = 90;
int obAng = 11;
int omegaD = 0;
byte freeSpace = 1;
byte pLearn = 0;

active proctype moving()
{
    do
    :: ((obDist > 30) && (freeSpace == 0)) ->
                                d_step{RESPOND_TO_OB_BY_TURNING();};
                                d_step{MOVE_FORWARD();};
                                d_step{LEARN();};
    :: ((obDist < 30) || (freeSpace == 1)) ->
                                atomic{GENERATE_NEW_OB();};
    od;
}
```

Figure 6.1: Promela code for the Relative model

plicit model, in Table 4.1). If there is a collision (`obDist<30`) then the agent turns $90°$, which is represented by a transition to a free-space state. From here the next obstacle to enter the COI arrives from a *legal* angle and distance –specified by the Agent-centric abstraction calculations. For the Relative model, this is anywhere along the front curve of the COI (see Figure 6.2). Note that it is also possible that no obstacle appears and that the agent moves forward into another free-space state.

6.1.1 Assumptions

The assumptions made about this system are the same as for the Explicit model and simulator, see Chapter 4. This environmental complexity is also the same, which fulfils assumption `A1` from the Agent-centric abstraction. That is, it only allows one obstacle in the COI at any given time. The exact parameters used for the COI representation are shown in Figure 5.9.

Figure 6.2, represents the specification we use for the COI in our Relative model. The antennas are separated by a $60°$ angle, measured from the centre of the robot, and are divided into six sections of 10 units per section; where the first 10 unit section (section 1, closest to the robot) represents the proximal antennas

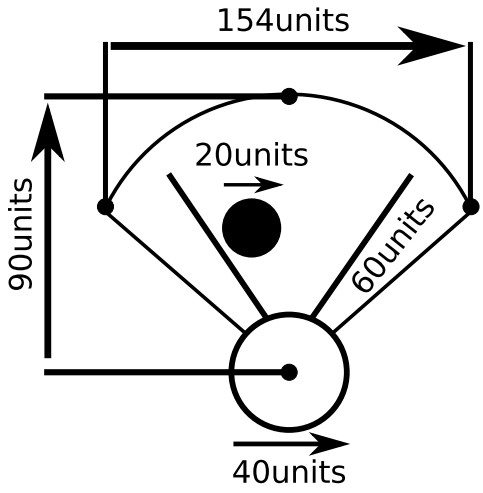


Figure 6.2: Cone of influence specification for the Relative model.

and the rest represent the distal antennas (sections $2 \rightarrow 6$). Note that the proximal antennas overlap with section 1 of the distal antennas and make contact instead of them; hence, section 1 of the distal antennas are omitted from our models.

The COI represents only the area of the system that we are concerned with, in regards to the type of properties to be checked. For this type of ABL system, we are concerned with properties that relate to learning and, consequentially, to collisions with obstacles. Therefore, this specification of the COI is sufficient for our Relative model.

The Relative model needs to fulfil all the assumptions for the Agent-centric abstraction (see Section 5.2). Hence, we apply Formulas 5.1 and 5.2 to our system's parameters in 6.1 and 6.2 respectively. (Note that in all the following calculations we only show up to 2 decimal places for brevity.)

$$\begin{aligned} \textit{Critical distance} &= 2 * (\tan(0.5 * 60) * (10 + 20)) = 34.64 \\ \\ \textit{Direct collision} &= \begin{cases} \texttt{true}, & \texttt{if}\,(34.64 > 20) \\ \texttt{false}, & \texttt{otherwise} \end{cases} \end{aligned} \tag{6.1}$$

$$
\begin{aligned}
MPL &= (10 \div (\sin(0.5 * 60))) - 10 = 10 \\
IPR &= \begin{cases} \texttt{true}, & \texttt{if}\,(10 \geq 10) \\ \texttt{false}, & \texttt{otherwise} \end{cases}
\end{aligned}
\tag{6.2}
$$

These calculations show that both *indefinite proximal reactions* (IPR) and *direct collisions* are possible for this specification. This is not an issue in our case, as ICO learning is not affected by these kind of collisions; hence assumption A2 holds. Next we calculate for the possibility of indirect crashing, with a view to fulfilling assumption A3. Hence, we apply Formulas 5.3 and 5.4 to our system's parameters in 6.3 and 6.4 respectively (abridged version of IPR calculation).

$$
\begin{aligned}
\texttt{Z} &= 90 - (60 \div 2) &&= 60 \\
\texttt{X} &= (10 + 20) * \tan(60) &&= 34.64 \\
\texttt{M} &= 10 + 1 &&= 11 \\
\texttt{N} &= 60 \div 2 &&= 30 \\
\texttt{Y} &= 11 \div (\sin(30)) &&= 22 \\
\\
(\texttt{Y} < \texttt{X}) &\therefore \texttt{PIC = true}
\end{aligned}
\tag{6.3}
$$

$$
\begin{aligned}
\texttt{P} &= \sqrt{(10+20)^2 + 34.64^2} &&= 45.82 \\
\texttt{L} &= \sqrt{22^2 - 11^2} &&= 19.05 \\
\texttt{V} &= 45.82 - 19.05 &&= 26.77 \\
\texttt{H} &= \sqrt{11^2 + 26.77^2} &&= 28.95 \\
\\
\texttt{MDBO} &= 28.95 + 20 + 60 + 10 &&= 118.95
\end{aligned}
$$

$$
\begin{aligned}
&\cdots \\
&\texttt{M} = 10 \\
&\cdots \\
&\texttt{IPR} = \begin{cases} \texttt{true}, & ((10+20) < 28.50 < (20+10)) \\ \texttt{false}, & \texttt{otherwise} \end{cases}
\end{aligned}
\tag{6.4}
$$

Property	Max ω_d	Stored states ($\times 10^5$)	Max search depth	Time (sec)
1P	6	121413	28881	0.32
2P		118129	28881	0.26

Table 6.1: Verification results for the Relative model.

Since the maximum distance between obstacles (that could cause an indirect collision) is less than the environmental complexity for our system (154), we do not consider indirect collisions for our Relative model. Additionally, the length of our proximal antenna is the same as the radius of an obstacle and therefore indefinite proximal reactions cannot occur either, and assumption `A3` also holds.

6.1.2 Verification

We check properties `1P` and `2P`, represented by Formula 4.6 and Formula 4.4 respectively (see Section 4.3).[10] The property `1P` is that the agent will eventually avoid all obstacles that it contacts with its distal antennas and, hence, has learnt successful distal avoidance behaviour. Property `2P` is that the agent will eventually be equal to, or less than its maximum level of learning (that learning stabilises). Note that the additional verifications to calculate the maximum level of learning are still required –here the maximum level is 6 ($\texttt{Max\_}\omega_d = 6$).

As with the Explicit models, the verifications are successful. In this case, there is only one verification required for each property –as opposed to running verifications on a set of explicit environments. The learning rate is again assumed to be 1. Results are given in Table 6.1.

6.1.3 Analysis

The verification of the properties for the Relative model required a larger state-space than with the Explicit model, but there is much more information derived. From property `1P` we can infer that in all variations of this type of environment, the agent eventually learns to stop all avoidable collisions. Hence, we assert that

[10] Verifying only properties `1P` and `2P` was sufficient to satisfy the aims of the researchers at the EEE, however verifying other properties is possible (see Chapter 7 for more explanation).

eventually it is always the case that if an agent detects an obstacle with a distal antenna it will avoid it.

Property `2P` is verified with a maximum learning level of 6; where this level applies to all variations of the environmental class. We infer from this that responding to obstacles to this extent (with this level of learning) is sufficient to avoid all collisions for this type of environment –collisions that are detectable with distal sensors.

The representation of an Explicit model provides an exact cut off point as to when an agent will finish learning for a particular environment. This precise evaluation is important when using a specific environment. However, the Relative model provides guarantees that have a much broader scope; i.e., wherever this system is deployed, if the environment fits the given specification, then we know that it will work correctly.

Chapter 7

Analysis and extensions

By applying model checking to ABL systems we have managed to improve the level of analysis that is commonly applied to them. Reasoning about properties over all possible paths in a system in one test provides a significant advantage over the testing of one path at a time. Additionally, exploiting the model checking framework further with the Agent-centric abstraction allows for more economic testing of a system: testing all possible environments in one model.

For our ABL systems the goal was to focus specifically on verifying properties that concerned the robot's learning. In the Explicit and Relative models we only considered the properties `1P` and `2P` (see Section 4.3.4). These properties were sufficient to satisfy the researchers from the EEE, as verifying these properties indicated the success and stability of the robot's learning. Although, our models are not restricted to these properties. For example, suppose that if the robot exceeds a given number of collisions it becomes inoperable. In this scenario we can consider properties of whether the agent's total number of collisions always stays within a given upper limit, or indeed always eventually reaches the given limit. Additionally, with the Explicit model we can consider positional properties of the robot. For example, verifying that an agent will never reach a specific location before passing through a given area. However, these types of additional properties are outside the scope of our research.

Model checking is surely a valuable technique in this field, yet we do not

advocate the use of model checking alone. We propose that model checking can be used alongside more traditional techniques, such as simulation; advocating model checking as an enhancement to the analysis of ABL systems, not as a replacement. In this chapter we begin by discussing some related work. Next, we compare model checking with simulation, and then go on to describe problems with our model checking approach and future enhancements to it.

7.1 Related work

There are several avenues of research which have some overlap with our analysis of ABL systems. Some of which we cover in this section, highlighting the differences with our approach.

The subject of motion planning for robots with similar goals to ours (such as obstacle avoidance) is covered in [79]. As with our work, the focus is on having formal guarantees that the continuous motion of a robot satisfies a specific temporal logic formula. However, in [79] the approach to calculating the motion of the robot is discretized such that the robot only has macro movements between cellular divisions of an environment. In addition, the position of the robot within an environment is of greater importance than with our models. Particularly, the specific sequence of movements is important (e.g., did the robot move around the environment in the correct order, from area `r1` to `r2`, to `r3`, and etc?).

Verification of an agent-based system is also considered in the domain of human-robot interaction in [80]. Here the scenario of robot helpers is discussed, where these systems are represented in the multi-agent modelling, simulation, and development environment, Brahms [81]. Like our approach, PROMELA and SPIN are used to formally verify system properties. However, in order to verify with SPIN an automatic converter from Brahms to PROMELA is used.

There has been considerable work done analysing MA systems by using Kripke modelling techniques. In [82] the behaviour of Unmanned Ariel Vehicles (UAVs) is formally analysed in the design phase through Kripke modelling and then model checking. Specifically, a group of UAVs are formalised with a Kripke model, then

properties of communication between the UAVs are expressed in temporal logic and verified using model checking tools.

In [83] and [84], multiple agents act as a swarm system where they are analysed in hierarchical layers of abstraction in order to use temporal logic to formally capture behavioural information (for a specific layer of abstraction). In the case study presented in [83], swarm robots navigate an environment via specific locations while avoiding: colliding with other swarm members, crashing into large polygonal obstacles, and moving outside a given area of a swarm cluster. These papers present a fully automated framework in which swarm-robot control-laws can be constructed, where controlling the essential features of a swarm is dealt with as a model checking problem.

Similar work on swarm systems uses model checking to verify whether given temporal logic properties are satisfied by all possible behaviours of a swarm [85]. Here the focus is on a particular swarm control algorithm which has been used and tested on real systems. The algorithm is refined using temporal analysis via model checking. This process of refinement involves iterating from highly abstract models to much more detailed models which are, ideally, to a level of detail equivalent to that of the real systems. Our research shares the concept of applying temporal analysis via model checking to an already existing system (using it as a reference point to assess results and refine models).

In [86] probabilistic model checking is applied to swarms systems. Model checking is proposed as an alternative to the common analysis of simulation –as we also advocate. PRISM is used to verify formulas relating to the behaviours of the swarms systems. Particularly, the behaviour of a foraging swarm colony is analysed, where $PCTL$ formulas are used to quantify the energy usage of the swarm over time in different scenarios. This is similar to the energy level quantification from our PRISM models (see Section 3.2.3).

Although much of this related work applies similar analysis and techniques that we use, it does not, however, completely overlap with our research. Much of the work agrees with our assertion of the common approach of simulation being insufficient to formally verify given properties of a system, and each also pro-

poses model checking as a potential solution. In our work we uniquely focus on the modelling of an unsupervised learning algorithm (ICO learning) as part of the agent in our model. We also utilise PROMELA's embedded C code to provided highly detailed models of our systems without producing intractable state-spaces. This removes the need to apply the methodology of using hierarchical levels of abstraction, or alternate specialised languages, in order to formally analyse properties of our ABL systems.

7.2 A note on polar coordinate representation

One of the main modelling choices for both the Agent-centric abstraction and the Explicit models was to use polar coordinates to express the location of obstacles and robots. This choice is made because of the type of property that is of interest: the robot's learning. By using this representation we are better able to express the precise turns of a robot when it encounters an obstacle, or another robot. Turns are symptomatic of the robot's learning itself; hence, expressing them with more accuracy allows for the level of learning to also be expressed more accurately.

A problem with using a polar coordinate representation is that as two lines diverge from the polar axis the angle between them remains the same while the distance between them increases. This becomes problematic when representing areas as polar coordinates. For example, as a robot is moved farther from the centre of the environment its area needs to be recalculated at every position of movement. We use a conversion to the Cartesian coordinate systems when recalculating areas, before translating back into polar coordinates.

Initially our Explicit model represented all areas as sets of coordinates. Now it is only the centre coordinates that are used in the state-space representation, while in the underlying embedded C code we preform the necessary calculations to generate the distances travelled and the areas occupied by the robot and obstacles. This allows us to preserve the angular information in the state-space, which we can then use to check properties, while maintaining accuracy in the mechanics of the model's transitions.

7.3 A note on PRISM

Although PRISM was only used in our preliminary models it proved an effective means of analysis for ABL systems.

The development of our PRISM models highlighted the type verification and analysis that can be done with this type of system. Additionally, the Learning obstacle avoidance PRISM model, in Section 3.2.3, showed the scope of our Agent-centric abstraction applications. Although this model was not generated from our Agent-centric abstraction, it did use a basic version of the COI by only considering interactions with the robot. It also had obstacles appear in a similar fashion as with the Relative model. With further development, full Agent-centric abstraction could be used to develop a more comprehensive PRISM model.

7.4 Comparison of classical closed-loop simulation and model checking methodologies

New strategies had to be developed to translate the behaviour-based approach into a form suitable for model checking. For simulation we used an existing framework to easily calculate the position of obstacles on the sensors, the new direction of the robot, and etc. In comparison, the PROMELA model was rather cumbersome, in that we had to construct a number of C code functions, in addition to just using pure PROMELA. However, we were able to adapt the code to represent the behaviour of the system. In order to simplify the PROMELA model we kept C code functions used for calculation hidden from the user (in included files). These functions can be reused in future models.

The advantage of the model-checking approach was that we could simply specify LTL properties to define behaviour that was expected for *all* paths for our model. That is, we did not have to run an exhaustive set of simulations to verify behaviour –the model checker would find *any* error path if it existed. In addition, our Relative model allowed us to check certain properties for all possible environments: if there was any distribution of obstacles for which one of our

properties did not hold, the model checker would find it. Having the capacity to examine error trails allowed us to not only debug our models, but to identify the pathological case in which one of the initial properties did not hold (i.e., the situation in which the robot hit an obstacle *head on*, without it first making contact with a distal sensor). This allowed us to strengthen the property to ignore this unusual case.

In addition, model checking allows us to identify deficiencies before, during, and after learning. That the robot cannot see obstacles which are hitting it head on is clearly a deficiency of its sensor distribution. While simple to spot in our example, more complex sensor motor setups will make it much more difficult to identify deficiencies which might occur only rarely. However unlikely, if these cases could cause damage to the robot or a deterioration its performance (say) then they need to be tackled appropriately. Model checking can help here (alongside classical simulation) to identify these problems in the design phase of a robot and will lead ultimately to a more reliable system.

The main drawback of the model checking approach is that it requires expert knowledge, both to construct a PROMELA specification with just the right level of abstraction, and to develop LTL properties to capture identified error behaviour. While the level of mathematical expertise required for our Explicit model is high, an even greater degree of theoretical knowledge is essential for the Relative model.

7.5 Model checking versus simulation for verification

By representing the same hardware system (see Section 2.2.3) for our simulator and our Explicit and Relative models we are able to compare the two approaches. Our system is simple, and subject to a number of assumptions. Indeed either approach requires assumptions to be made. The important issue is that the *same* assumptions are made in all cases, so that a fair comparison can be made. Our goal here is to illustrate the technique rather than to present a comprehensive suite of models.

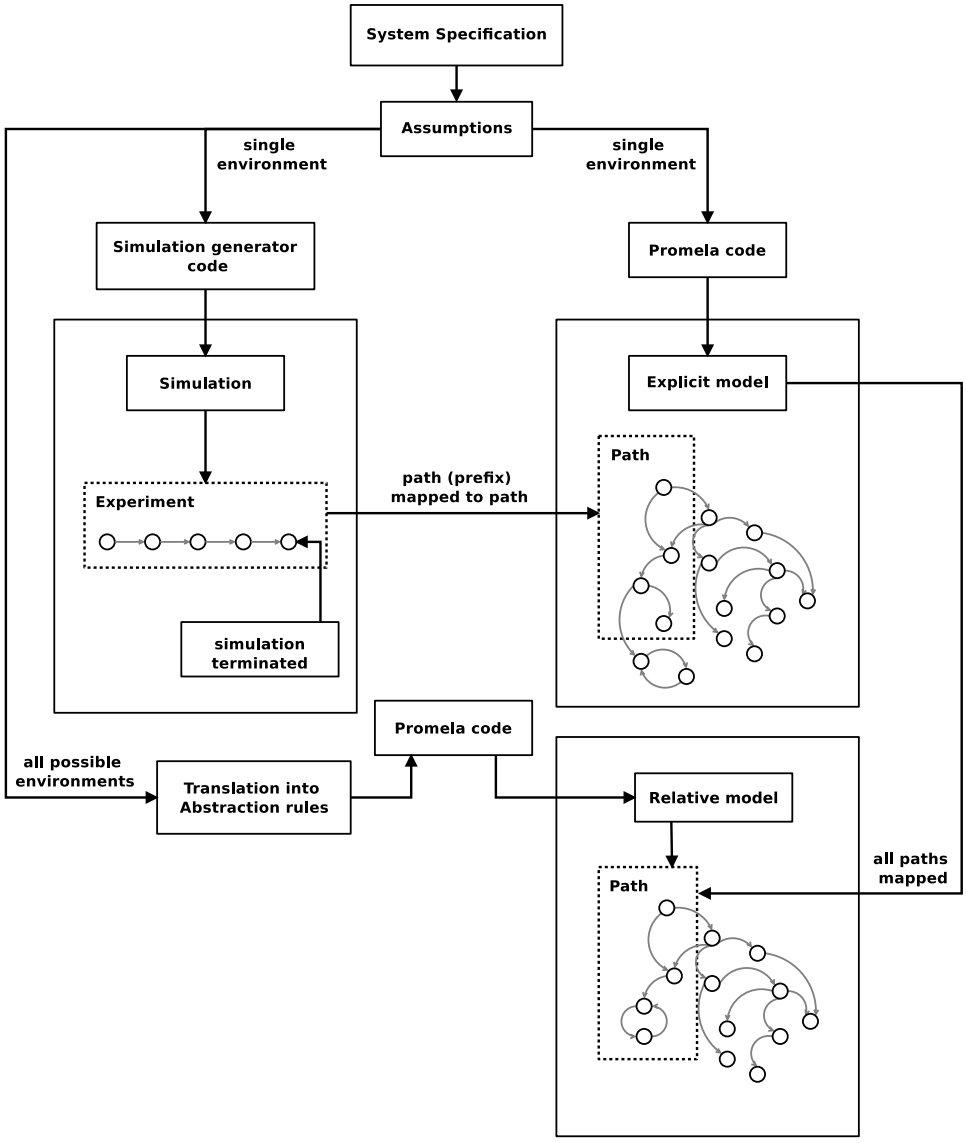


Figure 7.1: Comparison of approaches

Model checking involves analysis of a state-space. While in the classical simulation we could implement all variables as floating point (given analytical expressions of the entire environment), in model checking we need to discretise variables so that we can set up our state-space. Generally, the granularity of any discretisation of a robot simulation is determined by the signal to noise ratio of the sensors of the real robot and imperfections of its actuators [87, 88, 89]. This holds true for both classical simulation and model checking.

Simulation is equivalent to examining individual paths through a state-space. Exhaustive simulation (to cover all eventualities) is either very time-consuming or impossible. Model checking allows us to examine *all* possible paths, and to precisely express the property of interest (rather than relying on observation of simulation output). As illustrated in Figure 7.1, a single simulation is equivalent to a single path in our model. Often a simulation run is actually equivalent to a prefix of a path in our model (consisting of the first n states of the path, for some finite number n). A simulation is necessarily terminated at some point, whereas verification involves exploring all paths until either there are no further states, or until a cycle is detected.

Both simulation and model checking involve a degree of *abstraction*. The user of the technique must decide which aspect of the system to represent in their simulation/model. Our Explicit model is deliberately abstracted to the same degree as the simulation setup, so no additional information is lost. It is therefore straightforward to infer that we are modelling the *same thing* in each case. The power of model checking in this case is that, as described above, we can formally define a property and automatically check every path. Note that in the Explicit model there are few decision points (and so there are few paths), but in general a state-space contains many paths. For example, if there were multiple robots (as in the preliminary models Chapter 3), the ordering of steps taken by the different robots would lead to different paths, with different outcomes.

Having demonstrated the power of model checking with our Explicit model, we introduced the Relative model, which is a far more compact model, which not only merges symmetrically equivalent views (from the robot's perspective),

but combines equivalent environments into the same model. There are two major benefits to this Agent-centric abstraction. The specification of the Relative model is a much neater representation than the Explicit specification; e.g., fewer individual transitions need to be considered. In addition, results of verification hold for all environments (within a class of environment), which avoids the necessity of repeating the same experiments for similar, but different, environments. A drawback of the approach is that it requires expert knowledge of the system (e.g., intimate prior experience with the Explicit model). In addition, it differs so greatly from the physical system (and the simulation environment) that complex mathematical proof is required to ensure that the abstraction is *sound*; i.e., it preserves the properties in question.

7.6 Explicit model and Agent-centric abstraction: problems, improvements, and extensions

The main contribution of this thesis is the Agent-centric abstraction. However, in its current form it seems quite restrictive. In this section we describe the problems with developing a Relative model, and the improvements and additions that can be made to this abstraction that make it applicable to a much broader scope of ABL system.

The environments and robot behaviour that were represented in both our simulation setup and PROMELA specifications (with their associated models) were deliberately chosen to be simple. Whether creating a computer-based closed-loop simulation, or a PROMELA specification, it is necessary to make assumptions about the system that we are modelling. In either case we can not have limitless possibilities about the number of obstacles, or their shape. In addition we have to decide a priori whether to consider a fixed boundary, and, if so, the nature of the boundary.

The purpose of this work is to demonstrate the effectiveness of model checking (as a complementary approach to simulation), not to consider all possible environments or robot behaviour. In this section we discuss how we could adapt our

models to consider more complex scenarios. Note that, in all cases modifications are made to the PROMELA specification (SPIN produces the underlying models).

In each of the cases below, we assume that only the specified modification is to be implemented. Clearly we could combine the modifications in any way we like, but we only consider one at a time here, to make our explanation simpler. For each modification we first consider how the Explicit model's PROMELA specification would be adapted. The specification would be modified in much the same way as the simulation code would be modified. The corresponding Relative model's specification would, in all cases, require more detailed consideration.

When considering a new model we always start by using an Explicit model that is close to implementation level, and abstract from there (removing unnecessary variables, for example). Creating a Relative model requires experience of the Explicit model so as to gauge what the *equivalence classes* are. For example, in the Relative model considered in Chapter 6, the equivalence classes correspond to the possible positions of a single obstacle in the COI.

In each of the modifications listed below we indicate the corresponding equivalence class. Note that proof of soundness would involve proving that every state in a corresponding Explicit model maps to an equivalence class representative (and so to a state in the Relative model). We do not include all possible extensions here, just indicate a few that could be implemented very easily.

- **Inclusion of Environment Boundaries** Boundaries can easily be included in our Explicit model. In this case the boundary would be incorporated as a set of unreachable coordinates. The agent would respond to a signal from its sensors resulting from a collision with a boundary in the same way as it would a collision with an obstacle. Depending on the shape of the boundary (and assuming a single obstacle), the equivalence classes in the Relative model would correspond to the possible positions of an obstacle and a segment of boundary in the COI.
- **Arbitrary/Dynamic Boundaries** Any Explicit model would assume that a boundary was fixed. However, there is plenty of scope for allowing arbitrary boundary shapes, or dynamic boundaries in our Relative model, provided of course that

we assume (as we would do for simulation) that the possible types of boundary belong to a finite set. The equivalence classes in this case would be as for the previous example, but the number of possible different types of segment of visible boundary in the COI would increase.

- **Increased Complexity** This would mean allowing for there to be more than one obstacle within the COI at any time. The Explicit specification could be modified to accommodate this very easily (the array containing the positions of the obstacles would simply have to be altered). Assuming that there are at most N obstacles within the COI at any time, the equivalence classes (and hence the states in the Relative model) correspond to the possible positions of up to N obstacles within the COI.
- **Additional Robots** Our Explicit model involves a process definition of a robot, and a single instantiation of that process (agent). Adding additional robots would simply involve instantiating multiple agent processes (either with learning, or not). Our Relative model concerns the view *of a single robot*. Any additional robots would be viewed as dynamic obstacles. The behaviour of other robots (whether learning or not), would only be relevant within the COI (e.g., all possible movements of the other robots after a collision need to be considered).
- **Alternative Learning Algorithm** Both of our PROMELA specifications can be adapted very easily to accommodate an alternative learning algorithm. This would involve altering our C code functions determining the progress of the agent from any state after a collision. We could use our models to compare the consequences of different algorithms.
- **Dynamic Obstacles/Different Obstacles** By defining obstacles as processes, they could be defined as *dynamic*, either following a prescribed path, or following a nondeterministic path. Similarly, obstacles could be defined to have a variety of shapes and sizes, provided they can be defined and constrained before modelling. Dynamic obstacles, and different shapes and sizes of obstacle in the Relative model would require a minimal revision of the code (again requiring the different possibilities to be defined a priori). For example, dynamic obstacles would have a predefined behaviour when they were encountered in the COI

–this could be also be a nondeterministic behaviour.

- **Measuring explicit time** It is not possible to represent explicit time (for example to measure the amount of time between events) using SPIN alone, although the temporal ordering of events is clearly representable. When there is only one agent, there is a correlation between the number of global transitions between events, and the time between the events. It would therefore be possible to give a (discrete) representation of time using SPIN in this case. However, concurrent events are executed sequentially by SPIN, and so, when there is more than one component (i.e., agent) there is no such correlation. In order to prove quantitative properties, such as time between events, or the probability of an event, a more specialised model checker, such as the timed model checker Uppaal [90] or the probabilistic model checker PRISM [2] would be required. (In Section 3.2 we present PRISM models which apply probability and quantitative assessment to our systems.)

Chapter 8

Conclusion

We have presented various different approaches for applying model checking to ABL systems. The main objective was to provide an alternative to the prevailing methodology of simulation. Simulation may be cheap and easy to implement, but rigorous analysis is not possible without exhaustive simulation and physical experiments. In many of the ABL systems we examined, exhaustive simulation and experimental analysis are unfeasible at best, and impossible at worst. Through many examples of using model checking to represent and check various ABL systems, we present a strong case for its use; particularly as a tool to use alongside traditional approaches, opposed to a standalone paragon.

The development of the Agent-centric abstraction adds additional strength to our case for applying model checking. Its development was a result of wanting to formally analyse types of ABL system more efficiently. Through our definition and proof of a simulation relation we hope to persuade the reader of the soundness of this abstraction, and the scope of its application.

The Agent-centric abstraction provides an effective abstraction process for dealing with an entire class of systems in one model. The fundamental concept of this abstraction is widely applicable within the area of our systems. What we have presented here is a framework for the verification of properties in ABL systems. Additionally, we have highlighted how embedded C code functions can be utilised for modelling ABL systems with SPIN, by running them along side the PROMELA

model without adding to its state-space.

By proving the existence of a simulation relation between our Explicit model and Relative model, we show that it is possible to maintain the accuracy of the Explicit model while extending the scope to that of the Relative model. Our transition functions allow the abstraction framework to provide an automatic proof of a simulation relation. Hence, using our framework guarantees the soundness of the abstraction as part of its application.

The instantiation of our abstraction is presented in the Relative model. It provides a solid practical example of the effectiveness of our approach, which is strengthened through comparison with both Explicit models and simulation. Additions to our approach are describe in Chapter 7, where we give a better idea of the scope of Agent-centric abstraction.

A fundamental understanding of the hardware involved in our systems is essential to its modelling, we cover this information in Section 2.2.3. By breaking down the hardware interactions into a set of functions, we are able to abstract the representation of learning for our models. In both our Explicit and Relative models we demonstrate an effective representation of the ICO learning algorithm. This is achieved by abstracting the process of learning to just its trigger and behavioural response. Here, the trigger is the situation where a distal and subsequent proximal signal are correlated. The behavioural response is the increased sensitivity in the robots distal antennas.

We cover a broad background of literature in this area, with specific focus on the mathematical constructs associated with model checking. We also present a detailed description of both model checkers used in our practical work, including many of the model checking techniques available with them. Having an in depth knowledge of the tools at our disposal allowed for more effective models to be developed. For example, a full understanding of the embedded C code for PROMELA gave us much more accurate and powerful models. Additionally, the application of weak fairness in our multi-agent systems guaranteed that, for their verification, both agents were able to transition.

Throughout, we have shown the ways in which model checking can be applied

to ABL systems. By modelling existing systems like the ones from [77] we have learnt to recreate results that were obtained by simulation and experimentation. Our verification results for our Explicit and Relative models showed that even the state-spaces of our most complicated and accurate models can be tractable.

Principally, we have developed various models and techniques for modelling ABL systems, classifying the benefits and costs of each approach. We suggest that there is room for further development of our approach by expanding the scope of our Agent-centric abstraction and developing a custom-made tool to help automate the modelling of these systems.

8.1 Outstanding issues and implementations

While we have covered a broad application of model checking to ABL systems, including our Agent-centric abstraction with its proposed additions, one of the big future endeavours for this work is the development of a fully automated tool for dealing with this type of system.

We have already developed some automated code for the generation of environments for the Explicit model based on system parameters (see Appendix D). The combination of this with a model checking application seems feasible. This application would take in parameters of the obstacles and robots, the environmental complexity of the system, and perhaps also a learning algorithm to implement. From this it will be able to produce an Explicit model or a Relative model of the system.

There is also a much broader application for the basic representation of a robot, environment, and obstacles. For example, although the models would be geared toward obstacle avoidance, it is a simple change to represent obstacle attraction (by altering the polarity of ω_d, see Section 2.2.3). Additionally, fundamentally different types of model can be produced; e.g., by having obstacles represent something different, or altering the robot's method of learning. In fact, the scope of this framework can be exploited in many ways.

In addition to the auto-generation of an environment for an Explicit model,

this application could also include some visualisations of the system. We have already developed some auto-generation code of visualisations for models, using Gnuplot scripts (see Appendix D). These scripts can also be applied to a SPIN counterexample trace in order to generate a graphical representation of the precursor to a failed verification; i.e., the agent can be viewed as it fails to navigate its environment. (Note that the Figure 4.3 and Figure 4.5 were generated with our scrips.)

Another enhancement to this tool could be the inclusion of some form of automatic Theorem prover [91] into the application. This could aid in the proof of the simulation relation, or simply highlight the cases where a proof cannot be found.

Combining the visualisations with the auto-generated code and automated proof into a comprehensive tool would be one of the next stages in the advancement of this research.

Appendix A

PROMELA models

Here, we include the relevant code from the preliminary PROMELA and SPIN models.

A.1 Colliding robots

```
mtype = { clear, nogo};
byte roboX [2];
byte roboY [2];
bit validator = 0;
/*Each Robo sends its receive chan and the coords that it wants to test, to the environment*/
chan coordSend = [1] of { chan, byte, byte};
proctype robo(byte inX, inY, roboId) {
               chan thisFeeler = [1] of { mtype};
               byte curX = inX;
               byte curY = inY;
               byte futureX;
               byte futureY;
               mtype futureState = nogo;
reset:        futureX = curX; futureY = curY;
move:          do
               ::    futureY = futureY + 1; break;
               ::    futureY = futureY  1; break;
               ::    futureX = futureX  1; break;
               ::    futureX = futureX + 1; break;
               od;
testMove:  coordSend!thisFeeler, futureX, futureY;
wait:           thisFeeler?futureState;
               if
               :: (futureState == clear) > curX = futureX; curY = futureY;
                           roboX[roboId]=curX; roboY[roboId]=curY; goto move;
               :: (futureState == nogo) > goto reset;
               fi;
}
```

```
proctype environment() {
               byte posY;
               byte posX;
               byte count;
               chan sendRobo;
/*The "validator" is set to 1 here so that the validation test doesn't act on the arrays at
instantiation.*/
           validator = 1;
/*Reset "count" reset count and wait for new request from the robots.*/
waitTest: count = 0; coordSend? sendRobo, posX, posY;
           if
           :: posX > 2 > sendRobo!nogo; goto waitTest;
           :: posX < 0 > sendRobo!nogo; goto waitTest;
           :: posY > 2> sendRobo!nogo; goto waitTest;
           :: posY < 0 > sendRobo!nogo; goto waitTest;
           :: else > goto roboTest;
           fi;
roboTest: do
           ::          if
                       :: ((posX == roboX[count]) && (posY == roboY[count]))> sendRobo!nogo; goto
waitTest;
                       :: else >  if
                                   :: (count >= 1) > sendRobo!clear; goto waitTest;
                                   :: else > count = count + 1; goto roboTest;
                                   fi;
                       fi;
           od;
           goto waitTest;
}
init {
           roboX[0] = 1; roboY[0] = 2;
           roboX[1] = 2; roboY[1] = 3;
           run environment(); run robo(1, 2, 0); run robo(2, 3, 1);
}
```

A.2 Colliding robots verification output

```
pan: claim violated! (at depth 87)
pan: wrote pan_in.trail
pan: reducing search depth to 87
pan: end state in claim reached (at depth 87)
pan: wrote pan_in.trail
pan: reducing search depth to 86
(Spin Version 5.2.2  7 September 2009)
             + Partial Order Reduction
             + Compression
Full statespace search for:
             never claim             +
             assertion violations    + (if within scope of claim)
             acceptance   cycles     + (fairness disabled)
             invalid end states       (disabled by never claim)
Statevector 67 byte, depth reached 38049, errors: 8400
   249745 states, stored
 17913752 states, matched
 18163497 transitions (= stored+matched)
        0 atomic steps
hash conflicts:   7475575 (resolved)
Stats on memory usage (in Megabytes):
   20.721 equivalent memory usage for states (stored*(Statevector + overhead))
   10.235 actual memory usage for states (compression: 49.39%)
             statevector as stored = 23 byte + 20 byte overhead
    2.000 memory used for hash table (w19)
    0.003 memory used for DFS stack (m86)
   12.146 total actual memory usage
nr of templates: [ globals chans procs ]
collapse counts: [ 4886 487 211 8 2 ]
unreached in proctype robo
             line 37, "pan_in", state 26, "end"
             (1 of 26 states)
unreached in proctype environment
             line 52, "pan_in", state 8, "sendRobo!nogo"
           line 54, "pan_in", state 14, "sendRobo!nogo"
           line 68, "pan_in", state 38, "end"
           (3 of 38 states)
unreached in proctype :init:
          (0 of 8 states)
unreached in proctype :never:
          (0 of 8 states)
pan: elapsed time 608 seconds
pan: rate 411.06228 states/second
pan: avg transition delay 3.345e05 usec
174.24user 423.22system 10:07.52elapsed 98%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+3260048outputs (0major+3264minor)pagefaults 0swaps
```

A.3 Colliding robots (approaching-cell)

```
mtype = { clear, nogo};
byte roboX [2];
byte roboY [2];
byte roboAppX [2];
byte roboAppY [2];
bit validator = 0;
/*Each Robo sends its receive chan and the coords that it wants to test, to the environment*/
chan coordSend = [1] of { chan, byte, byte, byte};
proctype robo(byte roboID) {
               chan thisFeeler = [1] of { mtype};
               byte futureX;
               byte futureY;
               mtype futureState = nogo;
reset:         atomic { futureX = roboX[roboID]; futureY = roboY[roboID] };
move:          do
               ::    futureY = futureY + 1; break;
               ::    futureY = futureY  1; break;
               ::    futureX = futureX  1; break;
               ::    futureX = futureX + 1; break;
               od;
testApproach:                coordSend!thisFeeler, futureX, futureY, roboID;
waitApproach:                 thisFeeler?futureState;
               if
               :: (futureState == clear) > atomic { roboAppX[roboID] = futureX; roboAppY[roboID] =
futureY }; goto testMove;
               :: (futureState == nogo) > goto reset;
               fi;
testMove: coordSend!thisFeeler, futureX, futureY, roboID;
waitMove: thisFeeler?futureState;
               if
               :: (futureState == clear) > atomic { roboX[roboID]=futureX; roboY[roboID]=futureY };
goto move;
               :: (futureState == nogo) > atomic { roboAppX[roboID] = futureX;
roboAppY[roboID] = futureY }; goto reset;
               fi;
}
proctype environment() {
               byte posY;
               byte posX;
               byte count;
               byte roboID;
               chan sendRobo;
/*The "validator" is set to 1 here so that the validation test doesn't act on the arrays at
instantiation.*/
               validator = 1;
/*Reset "count" reset count and wait for new request from the robots.*/
waitTest: count = 0; coordSend? sendRobo, posX, posY, roboID;
               if
               :: posX > 2 > sendRobo!nogo; goto waitTest;
          :: posX < 0 > sendRobo!nogo; goto waitTest;
          :: posY > 2> sendRobo!nogo; goto waitTest;
          :: posY < 0 > sendRobo!nogo; goto waitTest;
          :: else > goto roboTest;
          fi;
roboTest: do
          ::          if
                      :: (count != roboID && (  ( (posX == roboX[count]) && (posY == roboY[count]) )
||
                            ( (posX==roboAppX[count]) && (posY==roboAppY[count]) )  ))> sendRobo!
nogo; goto waitTest;
                      :: else >  if
                                 :: (count >= 1) > sendRobo!clear; goto waitTest;
                                 :: else > count = count + 1; goto roboTest;
                                 fi;
                      fi;
          od;
          goto waitTest;
}
init {
          atomic { roboX[0] = 1; roboY[0] = 2; roboX[1] = 2; roboY[1] = 3 };
          run environment(); atomic { run robo(0); run robo(1) };
}
```

A.4 Colliding robots (approaching-cell) verification output

```
warning: for p.o. reduction to be valid the never claim must be stutterinvariant
(never claims generated from LTL formulae are stutterinvariant)
depth 0: Claim reached state 5 (line 19)
Depth=  411540 States=    1e+06 Transitions= 1.58e+06 Memory=   343.504 t=   3.11 R=   3e+05
Depth=  416528 States=    2e+06 Transitions= 3.28e+06 Memory=   378.465 t=    6.7 R=   3e+05
pan: resizing hashtable to w21..  done
(Spin Version 5.2.2  7 September 2009)
               + Partial Order Reduction
               + Compression
Full statespace search for:
               never claim          +
               assertion violations + (if within scope of claim)
               acceptance   cycles  + (fairness disabled)
               invalid end states    (disabled by never claim)
Statevector 63 byte, depth reached 416528, errors: 0
  2247944 states, stored
  1476357 states, matched
  3724301 transitions (= stored+matched)
        1 atomic steps
hash conflicts:   2988721 (resolved)
Stats on memory usage (in Megabytes):
  177.936  equivalent memory usage for states (stored*(Statevector + overhead))
   81.659  actual memory usage for states (compression: 45.89%)
           statevector as stored = 18 byte + 20 byte overhead
    8.000  memory used for hash table (w21)
  305.176  memory used for DFS stack (m10000000)
  394.668  total actual memory usage
nr of templates: [ globals chans procs ]
collapse counts: [ 43801 359 213 4 1 ]
unreached in proctype robo
           line 42, "pan_in", state 40, "end"
           (1 of 40 states)
unreached in proctype environment
           line 59, "pan_in", state 8, "sendRobo!nogo"
            line 61, "pan_in", state 14, "sendRobo!nogo"
            line 76, "pan_in", state 38, "end"
            (3 of 38 states)
unreached in proctype :init:
            (0 of 10 states)
unreached in proctype :never:
            line 24, "basicRobots1App.ltl", state 8, "end"
            (1 of 8 states)
pan: elapsed time 7.88 seconds
pan: rate 285272.08 states/second
pan: avg transition delay 2.1158e06 usec
7.86user 0.34system 0:08.20elapsed 99%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+8outputs (0major+101186minor)pagefaults 0swaps
```

A.5 Avoidance field robots

```
mtype = { clear, nogo};

byte roboX [2];
byte roboY [2];

byte roboDistalXBL [2];
byte roboDistalYBL [2];

bit validator = 0;

/*Each Robo sends its receive chan and the coords that it wants to test, to the environment*/
chan coordSend = [1] of { chan, byte, byte, byte};

proctype robo(byte inX, inY, roboId) {
        chan thisDistal = [1] of { mtype};

        byte curX = inX;
        byte curY = inY;
        byte roboID = roboId;
        byte futureX;
        byte futureY;
        mtype futureState = nogo;

reset:         atomic {futureX = curX; futureY = curY};
setDistal:  atomic {
                if
                :: (curX == 0) -> roboDistalXBL[roboID] = (curX);
                :: else -> roboDistalXBL[roboID] = (curX-1);
                fi;
                if
                :: (curY == 0) -> roboDistalYBL[roboID] = (curY);
                :: else -> roboDistalYBL[roboID] = (curY-1);
                fi
        };
move:   do
        ::    futureY = futureY + 1; break;
        ::    futureY = futureY - 1; break;
        ::    futureX = futureX - 1; break;
        ::    futureX = futureX + 1; break;
        od;

testMove: coordSend!thisDistal, futureX, futureY, roboID;

waitDistal: thisDistal?futureState;
        if
        :: (futureState == clear) -> atomic { curX = futureX; curY = futureY};
                atomic { roboX[roboID]=curX; roboY[roboID]=curY}; goto setDistal;
        :: (futureState == nogo) -> goto reset;
        fi;
}
```

```
proctype environment() {
        byte posY;
        byte posX;
        byte count;
        byte roboID;
        chan sendRobo;

/*The "validator" is set to 1 here so that the validation test doesn't act on the arrays at
instantiation.*/
        validator = 1;

/*Reset "count" reset count and wait for new request from the robots.*/
waitTest:       count = 0; coordSend? sendRobo, posX, posY, roboID;

/*The Robots environment is a grid (8x8), but the edging lines are considered as a wall*/
        if
        :: ((posX <= 7) && (posX >= 0) && (posY >= 0) && (posY <= 7))->  goto waitDistal;
        :: else -> sendRobo!nogo; goto waitTest;
        fi;

waitDistal:     count =0;

/*"count" only goes to '1' because there are only 2 Robots at the moment.*/
distalTest: do
          ::    if
                :: ((count <=1) && (count != roboID) && (posX>=roboDistalXBL[count]) &&
                    (posX <= (roboDistalXBL[count] + 2)) &&
                    (posY >= roboDistalYBL[count]) && (posY <= (roboDistalYBL[count] + 2))) ->
                        sendRobo!nogo; goto waitTest;
                :: else -> count = count+1;
                                if
                                :: (count <=1)-> goto distalTest;
                                :: (count >1) -> sendRobo!clear; goto waitTest;
                                fi;
                fi;
        od;
}

init {
        atomic { roboX[0] = 7; roboY[0] = 7; roboX[1] = 6; roboY[1] = 5; roboDistalXBL[0] = 6;
        roboDistalYBL[0] = 6; roboDistalXBL[1] = 5; roboDistalYBL[1] = 4};

        run environment(); atomic { run robo( roboX[0],  roboY[0], 0);
                                        run robo( roboX[1],  roboY[1], 1)};
}
```

A.6 Dual antenna robots (abridged code)

```
typedef Array { byte yAxis[22]; };
mtype = { clear, nogo};
Array xAxis[22];
byte roboDirection[2]; /*0 N, 1 NE, 2 E, 3 SE, 4 S, 5 SW, 6 W, 7 NW.*/
byte roboX[2];
byte roboY[2];
byte testV;
chan sensLongTest = [1] of { chan, chan, byte};
chan enviMove = [0] of { byte};

inline calcMove( iLeft, iRight) {
      errorVal = (iRight - iLeft);
      if
      :: (errorVal>0)->atomic{roboDirection=((roboDirection+1)%8)};goto move;
      :: (errorVal<0)->atomic{roboDirection=((roboDirection+7)%8)};goto move;
      :: else->goto move;
      fi; }

proctype environment() {
      byte newRoboX;
      byte newRoboY;
      byte roboID;

roboMove: enviMove?roboID;
atomic { newRoboX = roboX[roboID]; newRoboY = roboY[roboID]};

/*Moving the robot on the grid is an atomic, as the environment is not deemed to
need to calculate where the robot moves too.*/
atomic { if
      :: (roboDirection[roboID] == 0)->
        xAxis[newRoboX].yAxis[(newRoboY-1)] = 0;
xAxis[(newRoboX+1)].yAxis[(newRoboY-1)] = 0; /*Remove old pos.*/
        xAxis[newRoboX].yAxis[(newRoboY+1)] = 2;
xAxis[(newRoboX+1)].yAxis[(newRoboY+1)] = 2; /*Add new.*/
      :: (roboDirection[roboID] == 1)->
        xAxis[newRoboX].yAxis[(newRoboY-1)] = 0;
xAxis[(newRoboX-1)].yAxis[(newRoboY-1)] = 0;
        xAxis[(newRoboX-1)].yAxis[newRoboY] = 0; /*Remove old pos.*/
        xAxis[newRoboX].yAxis[(newRoboY+1)] = 2;
xAxis[(newRoboX+1)].yAxis[(newRoboY+1)] = 2;
        xAxis[(newRoboX+1)].yAxis[newRoboY] = 2; /*Add new pos.*/

      //Repeated for each direction...
      fi;};
goto roboMove;};

proctype robo(byte roboId) {
      chan feelerLeft = [1] of { byte};
      chan feelerRight = [1] of { byte};
      byte roboID = roboId;
      byte infoLeft;
      byte infoRight;
      byte errorVal;
      mtype futureState = nogo;

testMove: sensLongTest!feelerLeft, feelerRight, roboID;
feelerLeft?infoLeft;
feelerRight?infoRight;
calcMove( infoLeft, infoRight);
```

```
/*Reflex Sensor is combined with the movement phase.*/
move: if
:: (roboDirection[roboID] == 0)-> if
      :: ((xAxis[roboX[roboID]].yAxis[(roboY[roboID]+2)]!=0)||
      (xAxis[(roboX[roboID]+1)].yAxis[(roboY[roboID]+2)]!=0))->
            atomic{roboDirection=((roboDirection+1)%8)};goto move;/*Turn right*/
      :: else->atomic{roboY[roboID]=(roboY[roboID]+1)};enviMove!roboID;fi;
:: (roboDirection[roboID] == 1)-> if
      :: ((xAxis[(roboX[roboID]+1)].yAxis[(roboY[roboID]+2)]!=0)||
      (xAxis[(roboX[roboID]+2)].yAxis[(roboY[roboID]+1)]!= 0)||
      (xAxis[(roboX[roboID]+2)].yAxis[(roboY[roboID]+2)] == 1))->
            atomic{roboDirection=((roboDirection+1)%8)};goto move;/*Turn right*/
      :: else-> atomic {roboX[roboID] = (roboX[roboID] +1);
      roboY[roboID]=(roboY[roboID]+1)};enviMove!roboID;fi;

      //Repeated for each direction...
fi;
goto testMove;
}

proctype sensorLong() {
      byte posX;
      byte posY;
      byte roboID;
      byte thisRoboDire;
      chan roboLeft;
      chan roboRight;

waitTest: sensLongTest? roboLeft, roboRight, roboID;

atomic { posX = roboX[roboID]; posY = roboY[roboID]; }
thisRoboDire = roboDirection[roboID];

if
:: (thisRoboDire == 0)-> atomic { if
      :: (xAxis[ (posX-1) ].yAxis[ (posY+2) ]!= 0)->roboLeft!(2);
      :: else ->  if
            :: ( xAxis[ (posX-2) ].yAxis[ (posY+3) ]!= 0)->roboLeft!(4);
            :: else -> if
                  :: (xAxis[ (posX-3) ].yAxis[ (posY+4) ]!= 0)->roboLeft!(6);
                  :: else ->  if
                        :: (xAxis[(posX-4)].yAxis[(posY+5)]!= 0)->oboLeft!(8);
                        :: else -> roboLeft!(10); fi;
                  fi;
            fi;
      fi; };
      atomic { if
      :: (xAxis[(posX+2)].yAxis[(posY+2)]!= 0)-> roboRight!(2);goto waitTest;
            :: else ->  if
            :: ( xAxis[(posX+3)].yAxis[ (posY+3) ]!= 0)->roboRight!(4);
            :: else -> if
                  :: ( xAxis[(posX+4)].yAxis[ (posY+4) ]!= 0)->roboRight!(6);
                  :: else ->  if
                        :: (xAxis[(posX+5)].yAxis[(posY+5)]!= 0)->roboRight!(8);
                        :: else -> roboRight!(10); fi;
                  fi;
            fi;
      fi; };

      //Repeated for each direction...
fi;
goto waitTest; }
```

```
init {
atomic { /*Grid Walls*/
      xAxis[0].yAxis[0] = 1; xAxis[1].yAxis[0] = 1; xAxis[2].yAxis[0] = 1;
xAxis[3].yAxis[0] = 1; xAxis[4].yAxis[0] = 1;
      xAxis[5].yAxis[0] = 1; xAxis[6].yAxis[0] = 1; xAxis[7].yAxis[0] = 1;
xAxis[8].yAxis[0] = 1; xAxis[9].yAxis[0] = 1;
      xAxis[10].yAxis[0] = 1; xAxis[11].yAxis[0] = 1; xAxis[12].yAxis[0] = 1;
xAxis[13].yAxis[0] = 1; xAxis[14].yAxis[0] = 1;
      xAxis[15].yAxis[0] = 1; xAxis[16].yAxis[0] = 1; xAxis[17].yAxis[0] = 1;
xAxis[18].yAxis[0] = 1; xAxis[19].yAxis[0] = 1;
      xAxis[20].yAxis[0] = 1; xAxis[21].yAxis[0] = 1; xAxis[0].yAxis[1] = 1;
xAxis[0].yAxis[2] = 1; xAxis[0].yAxis[3] = 1;
      xAxis[0].yAxis[4] = 1; xAxis[0].yAxis[5] = 1; xAxis[0].yAxis[6] = 1;
xAxis[0].yAxis[7] = 1; xAxis[0].yAxis[8] = 1;
      xAxis[0].yAxis[9] = 1; xAxis[0].yAxis[10] = 1; xAxis[0].yAxis[11] = 1;
xAxis[0].yAxis[12] = 1; xAxis[0].yAxis[13] = 1;
      xAxis[0].yAxis[14] = 1; xAxis[0].yAxis[15] = 1; xAxis[0].yAxis[16] = 1;
xAxis[0].yAxis[17] = 1; xAxis[0].yAxis[18] = 1;
      xAxis[0].yAxis[19] = 1; xAxis[0].yAxis[20] = 1; xAxis[21].yAxis[1] = 1;
xAxis[21].yAxis[2] = 1; xAxis[21].yAxis[3] = 1;
      xAxis[21].yAxis[4] = 1; xAxis[21].yAxis[5] = 1; xAxis[21].yAxis[6] = 1;
xAxis[21].yAxis[7] = 1; xAxis[21].yAxis[8] = 1;
      xAxis[21].yAxis[9] = 1; xAxis[21].yAxis[10] = 1; xAxis[21].yAxis[11] = 1;
xAxis[21].yAxis[12] = 1; xAxis[21].yAxis[13] = 1;
      xAxis[21].yAxis[14] = 1; xAxis[21].yAxis[15] = 1; xAxis[21].yAxis[16] = 1;
xAxis[21].yAxis[17] = 1; xAxis[21].yAxis[18] = 1;
      xAxis[21].yAxis[19] = 1; xAxis[21].yAxis[20] = 1; xAxis[0].yAxis[21] = 1;
xAxis[1].yAxis[21] =1; xAxis[2].yAxis[21] = 1;
      xAxis[3].yAxis[21] = 1; xAxis[4].yAxis[21] = 1; xAxis[5].yAxis[21] = 1;
      xAxis[6].yAxis[21] = 1; xAxis[7].yAxis[21] = 1;
      xAxis[8].yAxis[21] = 1; xAxis[9].yAxis[21] = 1; xAxis[10].yAxis[21] = 1;
xAxis[11].yAxis[21] = 1; xAxis[12].yAxis[21] = 1;
      xAxis[13].yAxis[21] = 1; xAxis[14].yAxis[21] = 1; xAxis[15].yAxis[21] = 1;
xAxis[16].yAxis[21] = 1; xAxis[17].yAxis[21] = 1;
      xAxis[18].yAxis[21] = 1; xAxis[19].yAxis[21] = 1; xAxis[20].yAxis[21] = 1;
xAxis[21].yAxis[21] = 1;

/*Obstacles*/
/*xAxis[8].yAxis[20] =1; xAxis[8].yAxis[19] =1;*/

/*Robots*/
roboX[0] = 2; roboY[0] = 4;
xAxis[2].yAxis[4] = 2; xAxis[2].yAxis[5] = 2; xAxis[3].yAxis[4] = 2;
xAxis[3].yAxis[5] = 2;

roboX[1] = 14; roboY[1] = 10;
xAxis[14].yAxis[10] = 2; xAxis[14].yAxis[11] = 2; xAxis[15].yAxis[10] = 2;
xAxis[15].yAxis[11] = 2;};

testV = 5; run environment(); run sensorLong();
atomic {run robo(0); run robo(1)};
}
#include "newRobots2NoCrash.ltl"
```

Appendix B

PRISM models

Here, we include the relevant code from all the PRISM models.

B.1 Colliding robots (abridged code)

```
dtmc

// CONSTANTS
const int n; // size of the grid

formula S1up = min( min(1,max(n-y1,0)), max((x1=x2?0:1),((y1+1)=y2?0:1)) );
formula S1right = min( min(1,max(n-x1,0)), max((y1=y2?0:1),((x1+1)=x2?0:1)) );
formula S1down = min( min(1,max(y1-1,0)), max((x1=x2?0:1),((y1-1)=y2?0:1)) );
formula S1left = min( min(1,max(x1-1,0)), max((y1=y2?0:1),((x1-1)=x2?0:1)) );

formula S2up = min( min(1,max(n-y2,0)), max((x2=x1?0:1),((y2+1)=y1?0:1)) );
formula S2right = min( min(1,max(n-x2,0)), max((y2=y1?0:1),((x2+1)=x1?0:1)) );
formula S2down = min( min(1,max(y2-1,0)), max((x2=x1?0:1),((y2-1)=y1?0:1)) );
formula S2left = min( min(1,max(x2-1,0)), max((y2=y1?0:1),((x2-1)=x1?0:1)) );


module Robot1/Robot2
        dChoice :[0..3] init 0;
        x1 : [1..n] init 5;   // x position of robot.
        y1 : [1..n] init 5;   // y position of robot.
        apX1 : [1..n] init 5; // x position that the robot is going to move to.
        apY1 : [1..n] init 5; // y position that the robot is going to move to.

        [move1] (dChoice=0 & S1up=1 & !(y1=n)) -> 1.0 : (y1'=y1+1); // moves up
        [move1] (dChoice=0 & S1up=0) -> 0.5 : (dChoice'=1) + 0.5 : (dChoice'=3);
        [move1] (dChoice=1 & S1right=1 & !(x1=n)) -> 1.0 : (x1'=x1+1); // moves right
        [move1] (dChoice=1 & S1right=0) -> 0.5 : (dChoice'=2) + 0.5 : (dChoice'=0);
        [move1] (dChoice=2 & S1down=1 & !(y1=1)) -> 1.0 : (y1'=y1-1); // moves down
        [move1] (dChoice=2 & S1down=0) -> 0.5 : (dChoice'=3) + 0.5 : (dChoice'=1);
        [move1] (dChoice=3 & S1left=1 & !(x1=1)) -> 1.0 : (x1'=x1-1); // moves left
        [move1] (dChoice=3 & S1left=0) -> 0.5 : (dChoice'=0) + 0.5 : (dChoice'=2);
endmodule
```

B.2 Dual antenna robots (abridged code)

```
// GRID WORLD MODEL OF ROBOT
dtmc
// CONSTANTS
const int n; // size of the grid
//Agent 1
const int posX1; // x coordinate.
const int posY1; // y coordinate.
const int inD1;  // facing direction (8).
//Agent 2
const int posX2;
const int posY2;
const int inD2;

// Bracket Structure:
// max (a, b)  --a=-b
//     a=( (c6) (c5) (c4) (c3) (c2) (c1) ), b=( (c-6) (c-5) (c-4),(c-3),(c-2),(c-1) )
//             c6=max( (()&()?1:0),(()&()?1:0),(()&()?1:0),(()&()?1:0) )
//             c6: Check all 4 of the gird cells occupied by the other Robot
//
// All the c-'s are the left-had-side antenna sensor values and the other c's are the
// right-had-side values. The example below is based on the agent is facing North.
//
// Right
//c6: max(0,((x1+2)=x2)&((y1+2)=y2)?6:0, ((x1+2)=(x2+1))&((y1+2)=y2)?6:0,
//       ((x1+2)=x2)&((y1+2)=(y2+1))?6:0, ((x1+2)=(n+1))&((y1+2)=(n+1))?6:0)
//c5: max(0,((x1+3)=x2)&((y1+3)=y2)?5:0, ((x1+3)=(x2+1))&((y1+3)=y2)?5:0,
//       ((x1+3)=x2)&((y1+3)=(y2+1))?5:0, ((x1+3)=(x2+1))&((y1+3)=(y2+1))?5:0)
//c4: max(0,((x1+4)=x2)&((y1+4)=y2)?4:0, ((x1+4)=(x2+1))&((y1+4)=y2)?4:0,
//       ((x1+4)=x2)&((y1+4)=(y2+1))?4:0, ((x1+4)=(x2+1))&((y1+4)=(y2+1))?4:0)
//c3: max(0,((x1+5)=x2)&((y1+5)=y2)?3:0, ((x1+5)=(x2+1))&((y1+5)=y2)?3:0,
//       ((x1+5)=x2)&((y1+5)=(y2+1))?3:0, ((x1+5)=(x2+1))&((y1+5)=(y2+1))?3:0)
//c2: max(0,((x1+6)=x2)&((y1+6)=y2)?2:0, ((x1+6)=(x2+1))&((y1+6)=y2)?2:0,
//       ((x1+6)=x2)&((y1+6)=(y2+1))?2:0, ((x1+6)=(x2+1))&((y1+6)=(y2+1))?2:0)
//c1: max(0,((x1+7)=x2)&((y1+7)=y2)?1:0, ((x1+7)=(x2+1))&((y1+7)=y2)?1:0,
//       ((x1+7)=x2)&((y1+7)=(y2+1))?1:0, ((x1+7)=(x2+1))&((y1+7)=(y2+1))?1:0)
//
// Left
//c-6: max(0,((x1-1)=x2)&((y1+2)=y2)?6:0, ((x1-1)=(x2+1))&((y1+1)=y2)?6:0,
//        ((x1-1)=x2)&((y1+1)=(y2+1))?6:0, ((x1-1)=(x2+1))&((y1+1)=(y2+1))?6:0)
//c-5: max(0,((x1-2)=x2)&((y1+2)=y2)?5:0, ((x1-2)=(x2+1))&((y1+2)=y2)?5:0,
//       ((x1-2)=x2)&((y1+2)=(y2+1))?5:0, ((x1-2)=(x2+1))&((y1+2)=(y2+1))?5:0)
//c-4: max(0,((x1-1)=x2)&((y1+1)=y2)?4:0, ((x1-1)=(x2+1))&((y1+1)=y2)?4:0,
//        ((x1-1)=x2)&((y1+1)=(y2+1))?4:0, ((x1-1)=(x2+1))&((y1+1)=(y2+1))?4:0)
//c-3: max(0,((x1-2)=x2)&((y1+2)=y2)?3:0, ((x1-2)=(x2+1))&((y1+2)=y2)?3:0,
//        ((x1-2)=x2)&((y1+2)=(y2+1))?3:0, ((x1-2)=(x2+1))&((y1+2)=(y2+1))?3:0)
//c-2: max(0,((x1-3)=x2)&((y1+3)=y2)?2:0, ((x1-3)=(x2+1))&((y1+3)=y2)?2:0,
//        ((x1-3)=x2)&((y1+3)=(y2+1))?2:0, ((x1-3)=(x2+1))&((y1+3)=(y2+1))?2:0)
//c-1: max(0,((x1-4)=x2)&((y1+4)=y2)?1:0, ((x1-4)=(x2+1))&((y1+4)=y2)?1:0,
//        ((x1-4)=x2)&((y1+4)=(y2+1))?1:0, ((x1-4)=(x2+1))&((y1+4)=(y2+1))?1:0)
```

```
// Robot 1 // Or Robot 2
//-------------------------------------------------------------------------------
//Determine the signal difference in the antennas.
formula S1NAntSig =

max(0, max(0,((x1+2)=x2)&((y1+2)=y2)?6:0, ((x1+2)=(x2+1))&((y1+2)=y2)?6:0,
       ((x1+2)=x2)&((y1+2)=(y2+1))?6:0, ((x1+2)=(n+1))&((y1+2)=(n+1))?6:0)
       max(0,((x1+3)=x2)&((y1+3)=y2)?5:0, ((x1+3)=(x2+1))&((y1+3)=y2)?5:0,
       ((x1+3)=x2)&((y1+3)=(y2+1))?5:0, ((x1+3)=(x2+1))&((y1+3)=(y2+1))?5:0)
       max(0,((x1+4)=x2)&((y1+4)=y2)?4:0, ((x1+4)=(x2+1))&((y1+4)=y2)?4:0,
       ((x1+4)=x2)&((y1+4)=(y2+1))?4:0, ((x1+4)=(x2+1))&((y1+4)=(y2+1))?4:0)
       max(0,((x1+5)=x2)&((y1+5)=y2)?3:0, ((x1+5)=(x2+1))&((y1+5)=y2)?3:0,
       ((x1+5)=x2)&((y1+5)=(y2+1))?3:0, ((x1+5)=(x2+1))&((y1+5)=(y2+1))?3:0)
       max(0,((x1+6)=x2)&((y1+6)=y2)?2:0, ((x1+6)=(x2+1))&((y1+6)=y2)?2:0,
       ((x1+6)=x2)&((y1+6)=(y2+1))?2:0, ((x1+6)=(x2+1))&((y1+6)=(y2+1))?2:0)
       max(0,((x1+7)=x2)&((y1+7)=y2)?1:0, ((x1+7)=(x2+1))&((y1+7)=y2)?1:0,
       ((x1+7)=x2)&((y1+7)=(y2+1))?1:0, ((x1+7)=(x2+1))&((y1+7)=(y2+1))?1:0)  ) -
max(0, max(0,((x1-1)=x2)&((y1+2)=y2)?6:0, ((x1-1)=(x2+1))&((y1+1)=y2)?6:0,
       ((x1-1)=x2)&((y1+1)=(y2+1))?6:0, ((x1-1)=(x2+1))&((y1+1)=(y2+1))?6:0)
       max(0,((x1-2)=x2)&((y1+2)=y2)?5:0, ((x1-2)=(x2+1))&((y1+2)=y2)?5:0,
       ((x1-2)=x2)&((y1+2)=(y2+1))?5:0, ((x1-2)=(x2+1))&((y1+2)=(y2+1))?5:0)
       max(0,((x1-1)=x2)&((y1+1)=y2)?4:0, ((x1-1)=(x2+1))&((y1+1)=y2)?4:0,
       ((x1-1)=x2)&((y1+1)=(y2+1))?4:0, ((x1-1)=(x2+1))&((y1+1)=(y2+1))?4:0)
       max(0,((x1-2)=x2)&((y1+2)=y2)?3:0, ((x1-2)=(x2+1))&((y1+2)=y2)?3:0,
       ((x1-2)=x2)&((y1+2)=(y2+1))?3:0, ((x1-2)=(x2+1))&((y1+2)=(y2+1))?3:0)
       max(0,((x1-3)=x2)&((y1+3)=y2)?2:0, ((x1-3)=(x2+1))&((y1+3)=y2)?2:0,
       ((x1-3)=x2)&((y1+3)=(y2+1))?2:0, ((x1-3)=(x2+1))&((y1+3)=(y2+1))?2:0)
       max(0,((x1-4)=x2)&((y1+4)=y2)?1:0, ((x1-4)=(x2+1))&((y1+4)=y2)?1:0,
        ((x1-4)=x2)&((y1+4)=(y2+1))?1:0, ((x1-4)=(x2+1))&((y1+4)=(y2+1))?1:0) );

//Each other direction declared sepatately: formula S1NEAntSig ...and etc.

//Process for an agent (robot).
module Robot1
       d1 : [0..7] init inD1;
       x1 : [0..n] init posX1; // x position of robot
       y1 : [0..n] init posY1; // y position of robot

       [] (d1=0 & S1NAntSig=0 & !(y1=n)) -> 1.0 : (y1'=y1+1); //Moves N
       [] (d1=0 & S1NAntSig=0) ->  0.5 : (d1'=1) + 0.5 : (d1'=7);
       [] (d1=0 & S1NAntSig<0) -> 1.0 : (d1'=1);
       [] (d1=0 & S1NAntSig>0) -> 1.0 : (d1'=7);

       [] (d1=1 & S1NEAntSig=0 & !(y1=n) & !(x1=n)) -> 1.0:(y1'=y1+1)&(x1'=x1+1);//Moves NE
       [] (d1=1 & S1NEAntSig=0) -> 0.5 : (d1'=2) + 0.5 : (d1'=0);
       [] (d1=1 & S1NEAntSig<0) -> 1.0 : (d1'=2);
       [] (d1=1 & S1NEAntSig>0) -> 1.0 : (d1'=0);

       [] (d1=2 & S1EAntSig=0 & !(x1=n)) -> 1.0 : (x1'=x1+1); //Moves E
       [] (d1=2 & S1EAntSig=0) -> 0.5 : (d1'=3) + 0.5 : (d1'=1);
       [] (d1=2 & S1EAntSig<0) -> 1.0 : (d1'=3);
       [] (d1=2 & S1EAntSig>0) -> 1.0 : (d1'=1);

       [] (d1=3 & S1SEAntSig=0 & !(y1=0) & !(x1=n)) -> 1.0:(y1'=y1-1)&(x1'=x1+1);//Moves SE
       [] (d1=3 & S1SEAntSig=0) -> 0.5 : (d1'=4) + 0.5 : (d1'=2);
       [] (d1=3 & S1SEAntSig<0) -> 1.0 : (d1'=4);
       [] (d1=3 & S1SEAntSig>0) -> 1.0 : (d1'=2);

       [] (d1=4 & S1SAntSig=0 & !(y1=0)) -> 1.0 : (y1'=y1-1); //Moves S
       [] (d1=4 & S1SAntSig=0) -> 0.5 : (d1'=5) + 0.5 : (d1'=3);
       [] (d1=4 & S1SAntSig<0) -> 1.0 : (d1'=5);
       [] (d1=4 & S1SAntSig>0) -> 1.0 : (d1'=3);

       [] (d1=5 & S1SWAntSig=0 & !(y1=0) & !(x1=0)) -> 1.0:(y1'=y1-1)&(x1'=x1-1);//Moves SW
       [] (d1=5 & S1SWAntSig=0) -> 0.5 : (d1'=6) + 0.5 : (d1'=4);
       [] (d1=5 & S1SWAntSig<0) -> 1.0 : (d1'=6);
       [] (d1=5 & S1SWAntSig>0) -> 1.0 : (d1'=4);

       [] (d1=6 & S1WAntSig=0 & !(x1=0)) -> 1.0 : (x1'=x1-1); //Moves W
       [] (d1=6 & S1WAntSig=0) -> 0.5 : (d1'=7) + 0.5 : (d1'=5);
       [] (d1=6 & S1WAntSig<0) -> 1.0 : (d1'=7);
       [] (d1=6 & S1WAntSig>0) -> 1.0 : (d1'=5);

       [] (d1=7 & S1NWAntSig=0 & !(y1=n) & !(x1=0)) -> 1.0:(y1'=y1+1)&(x1'=x1-1);//Moves NW
       [] (d1=7 & S1NWAntSig=0) -> 0.5 : (d1'=0) + 0.5 : (d1'=6);
       [] (d1=7 & S1NWAntSig<0) -> 1.0 : (d1'=0);
       [] (d1=7 & S1NWAntSig>0) -> 1.0 : (d1'=6);
endmodule
```

B.3 Bean bag prediction

```
// Bean Bag Type Matching
dtmc

// Constants
//beanLim should be 1 more than the target number of beans.
const int beanLim;

//formulae
formula beanHalfLim = (beanLim/2);

module BeanBag

	s : [0..1] init 0;
	thisBean : [0..1] init 0;
	bPick : [0..1] init 1;
	beanNum : [0..100] init 0;
	beansSoFar : [0..beanLim] init 0;
	bagType : [0..4] init 0; // 1-5 are the different types of bag that can be chosen.
	choLim : [1..100] init 100;

	//bean is Blu 70% or the bean is Red 30%.
	[step] (bPick=1 & beanNum<100) -> 0.7 : (thisBean'=0)&(bPick'=0)&(beanNum'=beanNum+1) + 0.3 :
	(thisBean'=1)&(bPick'=0)&(beanNum'=beanNum+1);

	[step] (bPick=0 & beanNum<beanLim & beansSoFar<beanLim) -> 1.0 : (beansSoFar' = beansSoFar +
	thisBean) & (bPick'=1);

	//This is the decision algoritm. The hypotheses which are used to select the bag type.
	[step] (bPick=0 & s=0 & beanNum=beanLim & beansSoFar=0) -> 1.0 : (bagType' = 0) & (s'=1);

	[step] (bPick=0 & s=0 & beanNum=beanLim & beansSoFar<beanHalfLim &  beansSoFar>0)  ->
	1.0 : (bagType' = 1) & (s'=1);

	[step] (bPick=0 & s=0 & beanNum=beanLim & beansSoFar>=beanHalfLim &  beansSoFar<(beanLim-
	1)) -> 1.0 : (bagType' = 2) & (s'=1);

	[step] (bPick=0 & s=0 & beanNum=beanLim & beansSoFar=(beanLim-1)) -> 1.0 :  (bagType' = 3) &
	(s'=1);

endmodule
```

B.4 Learning obstacle avoidance

```
// Avoiding Obstacles
// This is similar to the bean bag examples except beans are now obstacles
// each obstacle requires a set amount of energy be avoided.

// We begin with only 3 turns: strong turn, weak turn, and collision (this requires the most turning energy).
// If a weak turn doesn't avoid an obstacle then a collision still occurs. Using the additional energy for a collision turn.

dtmc

// Constants
const int obLim;
const int angChoice;
//Pre-set Constants
const int collisionE = 9;
const int energyMAX = 900;
const int energyMAXSAFE = energyMAX - collisionE;

module ObAvoid

	s : [0..2] init 0;

	//Obstacles can either require 70deg or 30deg to avoid them. For Now.
	thisOb : [0..collisionE] init 0;
	obPick : [0..1] init 1;
	obNum  : [0..obLim] init 0;
	energy : [0..energyMAX] init 0;

	[step] (obPick=1 & obNum<obLim) -> 0.5 : (thisOb'=7)&(obPick'=0)&(obNum'=obNum+1) +
	0.5 : (thisOb'=3)&(obPick'=0)&(obNum'=obNum+1);

	[step] (obPick=0 & obNum<obLim & energy<energyMAXSAFE & thisOb>angChoice & s=0) ->
	1.0 : (energy' = energy + collisionE) & (s'=1);

	[step] (obPick=0 & obNum<obLim & energy<energyMAXSAFE & thisOb<=angChoice & s=0) ->
	1.0 : (energy' = energy + angChoice) & (obPick'=1);

	[step] (obPick=1 & obNum<obLim & energy<energyMAXSAFE & s=1) ->
	1.0 : (energy' = energy + angChoice) & (obPick'=1);

	[step] (obPick=0 & s=0 & (obNum>=obLim | energy>=energyMAX)) -> 1.0 : (s'=2);

endmodule
```

Appendix C

Explicit and Relative models

Here, we include the relevant code from the Explicit and Relative models.

C.1 Explicit model `Inline` and `Macros`

```
c_code {
        #define PI 3.141592653
        #define DEG 0.01745329
        #define ROBORAD 20
        #define OBRAD 10
        #define AAL 11
        #define AAR 69
        #define ADIST 8
        #define SENLEN 80
        #define LAMBDA 1
        #define OMEGAP 15
        #define ROBOMOVE 1
        #define ENVIRAD 200
        #define ENVIDIAM 400
        #define ANTLEN 60
        #define CRASHANGFROMCENTRE 9.825648155

        //GLOBALS//
        double x = 0;
        double prevX = 0;
        double roboA = 0;
        long double enviA = 0;
        long double enviD = 0;
        double obD = 0;
        double obA = 0;
        double relD = 0;
        double relA = 0;
        int moveDist = 1;
        int obDist = 0;
        int obAng = 0;
        int inCone = 0;
        int debug = 4;
        int testFlag = 0;
};

#define MOVE_FORWARD() { \
/*Declare Locals*/ \
if (now.relDist > 30) { \
        long double oZ = 0; \
        long double nZ = 0; \
        long double fZ = 0; \
        long double lOrg = 0; \
        long double hOrg = 0; \
        long double lNew = 0; \
        long double hNew = 0; \
        long double lFin = 0; \
        long double hFin = 0; \
        int oFR = 0; \
        int oFU = 0; \
        int nFR = 0; \
        int nFU = 0; \
\
```

```
        enviA, roboA, x, prevX);} \
        oZ = fmodl(enviA, (long double)90); \
        if ((enviA == 90) || (enviA ==270)) {lOrg = enviD; hOrg = 0; } \
        else if ((enviA == 0) || (enviA == 180)) {lOrg = 0; hOrg = enviD; } \
        else { \
                if ((enviA <=90) || ((enviA >=180)&&(enviA<=270))) { \
                        lOrg = (sin(oZ*DEG))*enviD; \
                        hOrg = (cos(oZ*DEG))*enviD; \
                } else { \
                        hOrg = (sin(oZ*DEG))*enviD; \
                        lOrg = (cos(oZ*DEG))*enviD; \
                } \
        } \
        nZ = fmod(roboA, 90.00);  \
        if ((roboA == 90) || (roboA ==270)) {lNew = moveDist; hNew = 0; } \
        else if ((roboA == 0) || (roboA == 180)) {lNew = 0; hNew = moveDist; } \
        else { \
                if ((roboA<90) || ((roboA>180)&&(roboA<270))) { \
                        lNew = (sin(nZ*DEG))*moveDist; \
                        hNew = (cos(nZ*DEG))*moveDist; \
                } else { \
                        hNew = (sin(nZ*DEG))*moveDist; \
                        lNew = (cos(nZ*DEG))*moveDist; \
                } \
        } \
        if ((enviA<180)&&(roboA>180) || (enviA>180)&&(roboA<180)) { lFin = fabs(lOrg - lNew);} \
        else {  lFin = lOrg + lNew;} \
        if ( (((enviA<90)||(enviA>270))&&((roboA>90)&&(roboA<270))) || \
        (((enviA>90)&&(enviA<270))&&((roboA<90)||(roboA>270))) ) {  \
                hFin = fabs(hOrg - hNew); \
        } else { hFin = hOrg + hNew; } \
        if ((hFin!=0)&&(lFin!=0)) { fZ = (atan(hFin/lFin)*(180/PI)); } \
        else { fZ = 0;} \
        enviD = sqrt((lFin*lFin)+(hFin*hFin)); \
        if ((enviA >=0) && (enviA <90))          { oFR = 1; oFU = 1;} \
        else if ((enviA >= 90) && (enviA <180))  { oFR = 1; oFU = 0;} \
        else if ((enviA >= 180) && (enviA <270)) { oFR = 0; oFU = 0;} \
        else if ((enviA >= 270) && (enviA <360)) { oFR = 0; oFU = 1;} \
        else {if (debug ==1){ Printf("FECKTAL ERROR with enviA = %Lf. \n", enviA);} oFR = 0; oFU = 0;}
\
        nFR = oFR; \
        nFU = oFU; \
        if ((oFR==1) && (oFU==1)) {  \
                if ((roboA>=180) && (roboA<360) && (lNew > lOrg)) { nFR = 0;} \
                if ((roboA>=90) && (roboA<270) && (hNew > hOrg))  { nFU = 0;} \
        } \
        else if ((oFR==1) && (oFU==0)) {  \
                if ((roboA>=180) && (roboA<360) && (lNew > lOrg)) { nFR = 0;} \
                if ( (((roboA>=270)&&(roboA<360))||((roboA>=0)&&(roboA<90))) && (hNew > hOrg)) { nFU =
1;} \
        } \
        else if ((oFR==0) && (oFU==0)) {  \
                if ((roboA>=0) && (roboA<180) && (lNew > lOrg))   { nFR = 1;} \
                if ( (((roboA>=270)&&(roboA<360))||((roboA>=0)&&(roboA<90))) && (hNew > hOrg)) { nFU =
1;} \
        } \
        else if ((oFR==0) && (oFU==1)) {  \
                if ((roboA>=0) && (roboA<180) && (lNew > lOrg))   { nFR = 1;} \
                if ((roboA>=90) && (roboA<270) && (hNew > hOrg))  { nFU = 0;} \
        } \
        /*Many catches for when the movement is along the axis lines/opposing them, \
        or when both are the same.*/ \
        if (roboA==enviA) { enviA = roboA;} \
        else if ((roboA==180)&&(enviA==0)&&(hNew>hOrg)) {enviA = 180;} \
        else if ((roboA==180)&&(enviA==0)&&(hNew < hOrg)) {enviA = 0;} \
        else if ((roboA==180)&&(enviA==0)&&(hNew == hOrg)) {enviA = 0;} \
        else if ((roboA==0)&&(enviA==180)&&(hNew>hOrg)) {enviA = 0;} \
        else if ((roboA==0)&&(enviA==180)&&(hNew < hOrg)) {enviA = 180;} \
```

```
        else if ((roboA==0)&&(enviA==180)&&(hNew == hOrg)) {enviA = 0;} \
        else if ((roboA==90)&&(enviA==270)&&(lNew>lOrg)) {enviA = 90;} \
        else if ((roboA==90)&&(enviA==270)&&(lNew < lOrg)) {enviA = 270;} \
        else if ((roboA==90)&&(enviA==270)&&(lNew == lOrg)) {enviA = 0;} \
        else if ((roboA==270)&&(enviA==90)&&(lNew>lOrg)) {enviA = 270;} \
        else if ((roboA==270)&&(enviA==90)&&(lNew < lOrg)) {enviA = 90;} \
        else if ((roboA==270)&&(enviA==90)&&(lNew == lOrg)) {enviA = 0;} \
        else if ((nFR==1)&&(nFU==1)) { enviA = 90 - fZ;} \
        else if ((nFR==1)&&(nFU==0)) { enviA = 90 + fZ;} \
        else if ((nFR==0)&&(nFU==0)) { enviA = 270 - fZ;} \
        else if ((nFR==0)&&(nFU==1)) { enviA = 270 + fZ;} \
        if (enviA>=360) {now.enviAng = 0;} \
        else {now.enviAng = ((int)(2*enviA)) - ((int)enviA);} \
        enviD = ((int)(2*enviD)) - ((int)enviD); \
        /*Don't want to wrap from a wrap.*/ \
        if ((enviD>=200) && (now.doWrap==0)) { enviD = 200;  now.doWrap=1;} \
        now.enviDist = (int)enviD; \
        } \
};


#define MOVE_ROBOT() c_code { \
/*Setting the roboA and the moveDist before MOVE_FORWARD() is called*/\
        roboA = (double) now.roboAng; \
        moveDist = ROBOMOVE; \
        MOVE_FORWARD(); \
};

#define GET_OB_REL_TO_ROBOT() {\
        /*Declare Locals*/ \
        long double oZ = 0; \
        long double nZ = 0; \
        long double fZ = 0; \
        long double lOrg = 0; \
        long double hOrg = 0; \
        long double lNew = 0; \
        long double hNew = 0; \
        long double lFin = 0; \
        long double hFin = 0; \
        int furtherRight = 0; \
        int furtherUp = 0; \
        double roboED = 0; \
        double roboEA = 0; \
        double roboFarAntiClock = 0; \
        double relAN = 0; \
\
        roboA = (double) now.roboAng; \
        roboED = (double)enviD; \
        roboEA = (double)enviA; \
        obD = (double)obDist; \
        obA = (double)obAng; \
        oZ = ((int)roboEA)%90; \
        if ((roboEA == 90) || (roboEA ==270)) {lOrg = roboED; hOrg = 0; } \
        else if ((roboEA == 0) || (roboEA == 180)) {lOrg = 0; hOrg = roboED; } \
        else { \
                if ((roboEA <=90) || ((roboEA >=180)&&(roboEA<=270))) { \
                        lOrg = (sin(oZ*DEG))*roboED; \
                        hOrg = (cos(oZ*DEG))*roboED; \
                } else { \
                        hOrg = (sin(oZ*DEG))*roboED; \
                        lOrg = (cos(oZ*DEG))*roboED; \
                } \
        } \
        nZ = ((int)obA)%90; \
        if ((obA == 90) || (obA ==270)) {lNew = obD; hNew = 0; } \
        else if ((obA == 0) || (obA == 180)) {lNew = 0; hNew = obD; } \
        else { \
                if ((obA<90) || ((obA>180)&&(obA<270))) { \
                        lNew = (sin(nZ*DEG))*obD; \
                        hNew = (cos(nZ*DEG))*obD; \
                } else { \
                        hNew = (sin(nZ*DEG))*obD; \
                        lNew = (cos(nZ*DEG))*obD; \
                } \
```

```
        } \
        if ((roboEA<180)&&(obA>180) || (roboEA>180)&&(obA<180)) { lFin = lOrg + lNew;} \
        else {  lFin = fabs(lOrg - lNew);} \
        if ( (((roboEA<90)||(roboEA>270))&&((obA>90)&&(obA<270))) || \
        (((roboEA>90)&&(roboEA<270))&&((obA<90)||(obA>270))) ) { \
                hFin = hOrg + hNew; \
        } else { hFin = fabs(hOrg - hNew);} \
        fZ = (atan(hFin/lFin)*(180/PI)); \
        if ((roboEA<=180)&&(obA<=180)) { if (lOrg > lNew) { furtherRight = 1;} else { furtherRight =
0;}} \
        else if ((roboEA>=180)&&(obA>=180)) { if (lOrg > lNew) { furtherRight = 0;} else
{ furtherRight = 1;}} \
        else if ((roboEA>180)&&(obA<180)) { furtherRight = 0;} \
        else if ((roboEA<180)&&(obA>180)) { furtherRight = 1;} \
        else {furtherRight = 0; furtherUp = 0;} \
        if (((roboEA<=90)||(roboEA>=270))&&((obA<=90)||(obA>=270))){ if(hOrg > hNew) { furtherUp=1;}
else { furtherUp=0;}} \
        else if (((roboEA>=90)&&(roboEA<=270))&&((obA>=90)&&(obA<=270))) { if(hOrg>hNew)
{ furtherUp=0;} else { furtherUp=1;}} \
        else if (((roboEA<=90)||(roboEA>=270)) && ((obA>=90)&&(obA<=270))) { furtherUp = 1;} \
        else if (((roboEA>=90)&&(roboEA<=270)) && ((obA<=90)||(obA>=270))) { furtherUp = 0;} \
        if ((furtherRight==1) && (furtherUp==1)) { relAN = 270 - fZ;} \
        else if ((furtherRight==1) && (furtherUp==0)) { relAN = 270 + fZ;} \
        else if ((furtherRight==0) && (furtherUp==1)) { relAN = 90 + fZ;} \
        else if ((furtherRight==0) && (furtherUp==0)) { relAN = 90 - fZ;} \
        roboFarAntiClock = ((((int)roboA) - 40)+360)%360; \
        relAN = ((int)(2*relAN)) - ((int)relAN); \
        relA = (((int)(relAN - roboFarAntiClock))+360)%360; \
        relD = sqrt((hFin*hFin) + (lFin*lFin)); \
        if ((relA <= 80)&&(relD <= 90)) { inCone = 1;} \
        else { inCone = 0;} \
        now.relDist = ((int)(2*relD)) - ((int)relD); \
};

#define RESPOND() c_code { \
        prevX = x; \
        now.prevSig = now.sig; \
        if (inCone==1) { \
                RESPOND_TO_OB_BY_TURNING(); \
                now.sig = (int)(fabs(x)); \
        } else { \
        /*If not inCone then we need to reset the signals*/ \
                x = 0; \
                now.sig = 0; \
        } \
}; \

#define RESPOND_TO_OB_BY_TURNING() { \
        /*Declare Locals*/ \
        double theta = 0; \
        double opp = 0; \
        double adj = 0; \
        double sAdj = 0; \
        double sHyp = 0; \
        int i = 0; \
        signed int flag = 0; \
        double contactPoint = 0; \
        double oAMAX = 0; \
        double lowR = 0; \
        double highR = 0; \
        double dOmegaD = 0; \
\
        roboA = (double)now.roboAng; \
        dOmegaD = (double)now.omegaD; \
        oAMAX = (atan((OBRAD/relD)))*(180/PI); \
        lowR = (relA - oAMAX); \
        if (lowR < 0) { lowR = 0;} \
        highR = (relA + oAMAX); \
        if (highR > 80) { highR = 80;} \
        if ( (AAL >= lowR) & (AAL <= highR) ) { theta = fabs(AAL - relA); flag = 1;} \
        else if ( (AAR >= lowR) & (AAR <= highR) ) { theta = fabs(AAR - relA); flag = -1;} \
        else {flag = 0;} \
        opp = relD*(sin(theta*(PI/180)));\
        if ((flag != 0 ) && (opp <= OBRAD)) { \
```

```
                adj = sqrt((relD*relD) - (opp*opp)); \
                if (adj <= 80) { \
                        for (i=0; i<=10; i++) {         \
                                sAdj = OBRAD - i; \
                                sHyp = sqrt((sAdj*sAdj) + (opp*opp));    \
                                if ((sHyp != 0) & (sHyp <= OBRAD)) {    \
                                        contactPoint = adj - sAdj;              \
                                        x = (flag)*(ADIST - ( ( contactPoint-(((int)contactPoint)%10)
)/10 )); \
                                        x = ((int)(2*x)) - ((int)x); \
                                        break; \
                                } \
                        } \
                } \
                else { x = 0;} \
        } \
        else { x = 0;}  \
        if ((x == 0) && (now.relDist <= 30)) { /*This is a crash*/ CRASH();} \
        else if (fabs(x) > 5) { roboA = (((int)(roboA + (x*OMEGAP)))+360)%360; LEARN_NOW();} \
        else { roboA = (((int)(roboA + (x*dOmegaD)))+360)%360;} \
        if (roboA>=360) { roboA=0;} \
        now.roboAng = (int)roboA; \
};

#define LEARN_NOW() { \
        if ( (prevX>=(-5)) && (prevX<=5) && (prevX!=0) ) { \
                now.omegaD = now.omegaD + LAMBDA;\
        } \
};

#define CRASH() { \
        /*Declare Locals*/ \
        double oZ = 0; \
        double oOpp = 0; \
        double oAdj = 0; \
        double oHyp = 0; \
        double nZ = 0; \
        double nOpp = 0; \
        double nAdj = 0; \
\
        /*If we crash with no proximal impact --We are inCone*/ \
        if ((relA==40) && (relD==30)) { \
                now.headOn = 1; /*Fixed Angle is 9.825648155*/ \
        } else if ((relA>=30)&&(relA<40)) { \
                oZ = relA - 11; \
                oHyp = 30; \
                oOpp = sin(oZ*DEG)*oHyp; \
                oAdj = sqrt((oHyp*oHyp)-(oOpp*oOpp)); \
                nOpp = oOpp - OBRAD; \
                nAdj = oAdj; \
                nZ = (atan(nOpp/nAdj))*(180/PI); \
                x = 6; \
                roboA = ((int)(roboA + (x*OMEGAP) + nZ))%360; \
        } else if ((relA<=50)&&(relA>40)) { \
                oZ = (relA - 40) - 11; \
                oHyp = 30; \
                oOpp = sin(oZ*DEG)*oHyp; \
                oAdj = sqrt((oHyp*oHyp)-(oOpp*oOpp)); \
                nOpp = oOpp - OBRAD; \
                nAdj = oAdj; \
                nZ = (atan(nOpp/nAdj))*(180/PI); \
                x = (-6); \
                roboA = (((int)(roboA + (x*OMEGAP) - nZ))+360)%360; \
        } \
        now.roboAng = ((int)(2*roboA)) - ((int)roboA); \
};

#define WRAP() c_code { \
        /*Declare Locals*/ \
        int l = 0; \
        long double orgEnviA = 0; \
        long double orgEnviD = 0; \
        int oppAng = 0; \
\
```

```
        roboA = (double)now.roboAng; /*We don't want to change roboAng.*/ \
        moveDist = ENVIDIAM; \
        oppAng = ((int)(roboA+180))%360; \
        roboA = oppAng; /*for MF function.*/ \
        orgEnviA = enviA; \
        orgEnviD = enviD; \
\
        MOVE_FORWARD(); \
        for (l=399; l--; l>=0) { \
        \
                if (enviD > 200) { \
                /*Reset the envi Values.*/ \
                        enviA = orgEnviA; \
                        enviD = orgEnviD; \
                        moveDist = l; \
                        MOVE_FORWARD(); \
                } else {break;} \
        } \
        roboA = (double)now.roboAng; /*Reset the roboAng.*/ \
        now.doWrap = 0; \
};

#define TRIVIAL_WRAP() c_code { \
                now.enviAng = (now.enviAng + 180)%360; \
                now.enviDist = 200; \
                enviA = ((int)(2*enviA)) - ((int)enviA); \
                enviA = ((int)enviA + 180)%360; \
                enviD = 200; \
                now.doWrap = 0; \
};

#define SCAN_APPROACHING_OBS() c_code { \
        /*Declare Locals*/ \
        int j = 0; \
        inCone = 0; \
        for (j=0; j<OBMAX; j++) { \
                obDist = now.arrObs[j].d; \
                obAng = now.arrObs[j].a; \
                GET_OB_REL_TO_ROBOT(); \
                if (inCone==1) {break;} \
        } \
};

inline HEAD_ON() {
/*Promela inline to add nondeterminism. Either we go right or we go left.*/
        if
        :: (headOn == 1) -> c_code{ Printf("HEADON1 \n");
                                x = 6;
                                now.sig = 6;
                                roboA = ((roboA+360)+ (x*OMEGAP) + CRASHANGFROMCENTRE);
                                roboA = ((int)(2*roboA)) - ((int)roboA);
                                roboA = ((int)roboA)%360; };
        :: (headOn == 1) -> c_code{ Printf("HEADON2 \n");
                                x = (-6);
                                now.sig = (-6);
                                roboA = ((roboA+360) - (6*OMEGAP) - CRASHANGFROMCENTRE);
                                roboA = ((int)(2*roboA)) - ((int)roboA);
                                roboA = ((int)roboA)%360; };
        fi;
        /*Now we assign the new value.*/
        c_code{ now.roboAng = ((int)(2*roboA)) - ((int)roboA); };
        headOn = 0;
};
```

C.2 Explicit model

```
/*Explicit Model: Using Functions (Macros)*/

c_decl { #include <math.h> }
#include "expInlineMacros.h"
#define OBMAX 4

/*Define a polar coordinate to be a distance and angle from origin (pole)*/
/*(Pole: centre of the environment. Polar axis: vertical line directed north.)*/
typedef polarCoord {int d; int a};

/*Setting the C_Track variables*/
c_track"&x" "sizeof(double)"
c_track "&prevX" "sizeof(double)"

c_track "&roboA" "sizeof(double)"
c_track "&enviD" "sizeof(long double)"
c_track "&enviA" "sizeof(long double)"

c_track "&obD" "sizeof(double)"
c_track "&obA" "sizeof(double)"
c_track "&relD" "sizeof(double)"
c_track "&relA" "sizeof(double)"

c_track "&obDist" "sizeof(int)"
c_track "&obAng" "sizeof(int)"
c_track "&moveDist" "sizeof(int)"
c_track "&inCone" "sizeof(int)"

/*Array of obstacles in the fixed environment*/
polarCoord arrObs[OBMAX];

/*Robot is initiated in the centre of the environment*/
int roboAng;
int enviDist;
int enviAng ;
int omegaD;
int sig;
int prevSig;
byte doWrap;
byte headOn;
int relDist;
```

```
proctype robot() {
        do
        :: (doWrap==0) ->       d_step{ SCAN_APPROACHING_OBS();
                                        RESPOND();
                                        MOVE_ROBOT();
                                        HEAD_ON();
                                };
        :: (doWrap==1) ->       d_step{ WRAP();
                                };
        od;
};


init {
        d_step{
                /* Set up the polar coordinates of the obstacles - fixed for model */
                /*(This setup will be automated with a C function.)*/
                arrObs[0].d = 122; arrObs[0].a = 350;
                arrObs[1].d = 121; arrObs[1].a = 69;
                arrObs[2].d = 121; arrObs[2].a = 149;
                arrObs[3].d = 121; arrObs[3].a = 229;

                roboAng = 0;
                enviDist = 0;
                enviAng = 0;
                omegaD = 0;
                doWrap = 0;
                sig = 0;
                prevSig = 0;
                headOn = 0;
                relDist = 200;

                c_code{
                        roboA = 0;
                        enviA = 0;
                        enviD = 0;
                };
        };
        atomic{ run robot();};
};
```

C.3 Relative model `Inline` and `Macros`

```
c_code {
        #define PI 3.14159265
        #define ROBORAD 20
        #define OBRAD 10
        #define AAL 11
        #define AAR 69
        #define ADIST 8
        #define SENLEN 80
        #define LAMBDA 1
        #define OMEGAP 15
        double obD, obA, theta, opp, adj, sAdj, hyp, sHyp, x,
         prevX, dOmegaD, oAMAX, lowR, highR, contactPoint = 0;
        int i, testDec  = 0;
        signed int flag = 0;
};

#define MOVE_FORWARD() c_code { \
        obD = (double) now.obDist; \
        obA = (double) now.obAng; \
        theta = fabs(obA - 40);  \
        adj = ( obD*cos(theta*(PI/180)) ); \
        opp = sqrt( (obD*obD) - (adj*adj) ); \
        adj = (adj - 1); \
        obD = sqrt( (opp*opp) + (adj*adj) ); \
        theta = (atan(opp/adj))*(180/PI); \
        if (obA < 40) { obA = 40 - theta; } \
        else { obA = 40 + theta; } \
        if ((obA < 0) | (obA >= 80)){ obA = 0; obD = 90; now.freeSpace = 1;} \
        now.obDist = ((int)(2*obD)) - ((int)obD); \
        now.obAng = ((int)(2*obA)) - ((int)obA); \
};

#define RESPOND_TO_OB_BY_TURNING() c_code { \
        obD = (double)now.obDist; \
        obA = (double)now.obAng; \
        dOmegaD = (double)now.omegaD; \
        oAMAX = (atan((OBRAD/obD)))*(180/PI); \
        lowR = (obA - oAMAX); \
        if (lowR < 0) { lowR = 0;} \
        highR = (obA + oAMAX); \
        if (highR > 80) { highR = 80;} \
        if ( (AAL >= lowR) & (AAL <= highR) ) { theta = fabs(AAL - obA); flag = -1;} \
        else if ( (AAR >= lowR) & (AAR <= highR) ) { theta = fabs(AAR - obA); flag = 1;} \
        else {flag = 0;} \
        opp = obD*(sin(theta*(PI/180)));\
```

```
        if ((flag != 0 ) && (opp <= OBRAD)) { \
                adj = sqrt((obD*obD) - (opp*opp)); \
                if (adj <= 80) { \
                        for (i=0; i<=10; i++) {         \
                                sAdj = OBRAD - i; \
                                sHyp = sqrt((sAdj*sAdj) + (opp*opp));    \
                                if ((sHyp != 0) & (sHyp <= OBRAD)) {    \
                                        contactPoint = adj - sAdj;              \
                                        x = (flag)*(ADIST - ( ( contactPoint-(((int)contactPoint)%10)
)/10 )); \
                                        x = ((int)(2*x)) - ((int)x); \
                                        break; \
                                } \
                        } \
                } \
                else { x = 0;} \
        } \
        else { x = 0;}  \
        if (fabs(x) > 5) { \
                obA = obA + (x*OMEGAP); \
                now.pLearn = 1; \
        } \
        else { obA = obA + (x*dOmegaD);} \
        if ((obA < 0) | (obA >= 80)) { now.freeSpace = 1;} \
        else { now.obAng = ((int)(2*obA)) - ((int)obA);} \
        now.sig = (int)(fabs(x)); \
};

#define LEARN() c_code { \
        if ((now.pLearn == 1) && (fabs(prevX) > 0) && (fabs(prevX) <= 5)) { \
                now.omegaD = now.omegaD + LAMBDA; \
                now.freeSpace = 1; \
        } \
        now.pLearn = 0; \
        prevX = fabs(x); \
        now.prevSig = (int)(fabs(prevX)); \
};

inline GENERATE_NEW_OB()  {
                obDist = 90;
                freeSpace = 0;
                c_code {
                        x=0;
                        prevX = 0;
                };

                do
                :: atomic { obDist = 90; obAng=1;}; break;
                :: atomic { obDist = 90; obAng=2;}; break;
                :: atomic { obDist = 90; obAng=3;}; break;
                :: atomic { obDist = 90; obAng=4;}; break;
                :: atomic { obDist = 90; obAng=5;}; break;
                :: atomic { obDist = 90; obAng=6;}; break;
                :: atomic { obDist = 90; obAng=7;}; break;
                :: atomic { obDist = 90; obAng=8;}; break;
                :: atomic { obDist = 90; obAng=9;}; break;
                :: atomic { obDist = 90; obAng=10;}; break;
                :: atomic { obDist = 90; obAng=11;}; break;
                :: atomic { obDist = 90; obAng=12;}; break;
                :: atomic { obDist = 90; obAng=13;}; break;
                :: atomic { obDist = 90; obAng=14;}; break;
                :: atomic { obDist = 90; obAng=15;}; break;
                :: atomic { obDist = 90; obAng=16;}; break;
                :: atomic { obDist = 90; obAng=17;}; break;
                :: atomic { obDist = 90; obAng=18;}; break;
                :: atomic { obDist = 90; obAng=19;}; break;
                :: atomic { obDist = 90; obAng=20;}; break;
                :: atomic { obDist = 90; obAng=21;}; break;
                :: atomic { obDist = 90; obAng=22;}; break;
                :: atomic { obDist = 90; obAng=23;}; break;
                :: atomic { obDist = 90; obAng=24;}; break;
                :: atomic { obDist = 90; obAng=25;}; break;
                :: atomic { obDist = 90; obAng=26;}; break;
                :: atomic { obDist = 90; obAng=27;}; break;
```

```
        :: atomic { obDist = 90; obAng=28;}; break;
        :: atomic { obDist = 90; obAng=29;}; break;
        :: atomic { obDist = 90; obAng=30;}; break;
        :: atomic { obDist = 90; obAng=31;}; break;
        :: atomic { obDist = 90; obAng=32;}; break;
        :: atomic { obDist = 90; obAng=33;}; break;
        :: atomic { obDist = 90; obAng=34;}; break;
        :: atomic { obDist = 90; obAng=35;}; break;
        :: atomic { obDist = 90; obAng=36;}; break;
        :: atomic { obDist = 90; obAng=37;}; break;
        :: atomic { obDist = 90; obAng=38;}; break;
        :: atomic { obDist = 90; obAng=39;}; break;
        :: atomic { obDist = 90; obAng=40;}; break;
        :: atomic { obDist = 90; obAng=41;}; break;
        :: atomic { obDist = 90; obAng=42;}; break;
        :: atomic { obDist = 90; obAng=43;}; break;
        :: atomic { obDist = 90; obAng=44;}; break;
        :: atomic { obDist = 90; obAng=45;}; break;
        :: atomic { obDist = 90; obAng=46;}; break;
        :: atomic { obDist = 90; obAng=47;}; break;
        :: atomic { obDist = 90; obAng=48;}; break;
        :: atomic { obDist = 90; obAng=49;}; break;
        :: atomic { obDist = 90; obAng=50;}; break;
        :: atomic { obDist = 90; obAng=51;}; break;
        :: atomic { obDist = 90; obAng=52;}; break;
        :: atomic { obDist = 90; obAng=53;}; break;
        :: atomic { obDist = 90; obAng=54;}; break;
        :: atomic { obDist = 90; obAng=55;}; break;
        :: atomic { obDist = 90; obAng=56;}; break;
        :: atomic { obDist = 90; obAng=57;}; break;
        :: atomic { obDist = 90; obAng=58;}; break;
        :: atomic { obDist = 90; obAng=59;}; break;
        :: atomic { obDist = 90; obAng=60;}; break;
        :: atomic { obDist = 90; obAng=61;}; break;
        :: atomic { obDist = 90; obAng=62;}; break;
        :: atomic { obDist = 90; obAng=63;}; break;
        :: atomic { obDist = 90; obAng=64;}; break;
        :: atomic { obDist = 90; obAng=65;}; break;
        :: atomic { obDist = 90; obAng=66;}; break;
        :: atomic { obDist = 90; obAng=67;}; break;
        :: atomic { obDist = 90; obAng=68;}; break;
        :: atomic { obDist = 90; obAng=69;}; break;
        :: atomic { obDist = 90; obAng=70;}; break;
        :: atomic { obDist = 90; obAng=71;}; break;
        :: atomic { obDist = 90; obAng=72;}; break;
        :: atomic { obDist = 90; obAng=73;}; break;
        :: atomic { obDist = 90; obAng=74;}; break;
        :: atomic { obDist = 90; obAng=75;}; break;
        :: atomic { obDist = 90; obAng=76;}; break;
        :: atomic { obDist = 90; obAng=77;}; break;
        :: atomic { obDist = 90; obAng=78;}; break;
        :: atomic { obDist = 90; obAng=79;}; break;
        :: atomic { obDist = 90; obAng=80;}; break;
        :: atomic { freeSpace = 1;};         break;
        od;
};
```

C.4 Relative model

```
/*Relative Model: Using Functions (Macros)*/

c_decl { #include <math.h> }
#include "relInlineMacros.h"

int obDist = 90;
int obAng = 11;
int omegaD = 0;
byte freeSpace = 1;
byte pLearn = 0;
int prevSig = 0;
int sig = 0;

/*Setting the C_Track variables*/
c_track "&x" "sizeof(double)"
c_track "&prevX" "sizeof(double)"
c_track "&obD" "sizeof(double)"
c_track "&obA" "sizeof(double)"
c_track "&dOmegaD" "sizeof(double)"

active proctype moving()
{
        do
        :: ((obDist > 30) && (freeSpace == 0)) -> d_step{RESPOND_TO_OB_BY_TURNING();};
                                                  d_step{MOVE_FORWARD(); LEARN();};
        :: ((obDist <= 30) || (freeSpace == 1)) -> atomic{GENERATE_NEW_OB();};
        od;
}
```

Appendix D

Basic auto-generation code

Here, we include some additional code based on our ideas for future work. The shape generation code is used for drawing obstacles and the robot. The line generation code is used for drawing sensor and boundaries.

D.1 Gnuplot shape generation `H` code

```
#define PI 3.14159265
#define DEG 0.01745329
#define OBMINDIST 155
#define OBMINDIST 155

typedef struct {
        int d; /*Distance*/
        int a; /*Angle*/
} coord_t;
/*Array of the Obstacles*/
coord_t arrObs[4];

coord_t setCoord(int dist, int angle) {
        coord_t rtn;
        rtn.d = dist;
        rtn.a = angle;
        return rtn;
}


coord_t genOb(coord_t otherOb, int angFrom) {

        double roboA, enviD, enviA = 0;
        double oZ, nZ, fZ, lOrg, hOrg, lNew, hNew, lFin, hFin = 0;
        int oFR, oFU, nFR, nFU = 0;
        int i = 1;

        coord_t rtnOb;
        rtnOb.a = 0;
        rtnOb.d = 0;

        /*Variables for the C code calculations (convert back to "int"s at the end)*/
        roboA = (double) angFrom; /*Angle ob is to previous one*/
        enviD = (double) otherOb.d;
        enviA = (double) otherOb.a;

        /*Triangle Original*/
        oZ = ((int)enviA)%90;
        if ((enviA == 90) || (enviA ==270)) {lOrg = enviD; hOrg = 0; }
        else if ((enviA == 180) || (enviA == 0)) {lOrg = 0; hOrg = enviD; }
```

```
	else {
		if ((enviA <=90) || ((enviA >=180)&&(enviA<=270))) {
			lOrg = (sin(oZ*DEG))*enviD;
			hOrg = (cos(oZ*DEG))*enviD;
		} else {
			hOrg = (sin(oZ*DEG))*enviD;
			lOrg = (cos(oZ*DEG))*enviD;
		}
	}
	/*Triangle New*/
	nZ = ((int)roboA)%90;
	if ((roboA == 90) || (roboA ==270)) {lNew = OBMINDIST; hNew = 0; }
	else if ((roboA == 180) || (roboA == 0)) {lNew = 0; hNew = OBMINDIST; }
	else {
		if ((roboA <=90) || ((roboA >=180)&&(roboA<=270))) {
			lNew = (sin(nZ*DEG))*OBMINDIST;
			hNew = (cos(nZ*DEG))*OBMINDIST;
		} else {
			hNew = (sin(nZ*DEG))*OBMINDIST;
			lNew = (cos(nZ*DEG))*OBMINDIST;
		}
	}
	/*Triangle Final*/
	if ((enviA<180)&&(roboA>180) || (enviA>180)&&(roboA<180)) { lFin = fabs(lOrg - lNew);}
	else {  lFin = lOrg + lNew;}

	if ( (((enviA<90)||(enviA>270))&&((roboA>90)&&(roboA<270))) ||
	(((enviA>90)&&(enviA<270))&&((roboA<90)||(roboA>270))) ) {
		hFin = fabs(hOrg - hNew);
	} else { hFin = hOrg + hNew; }

	if ((hFin!=0)&&(lFin!=0)) { fZ = (atan(hFin/lFin)*(180/PI)); }
	else { fZ = 0;}

	/*The new enviDist for the robot's new position*/
	enviD = sqrt((lFin*lFin)+(hFin*hFin));

	/*Set up original Quadrant*/
	if ((enviA >=0) && (enviA <90))          { oFR = 1; oFU = 1;}
	else if ((enviA >= 90) && (enviA <180))  { oFR = 1; oFU = 0;}
	else if ((enviA >= 180) && (enviA <270)) { oFR = 0; oFU = 0;}
	else if ((enviA >= 270) && (enviA <360)) { oFR = 0; oFU = 1;}

	/*Assign default values incare we do not leave the current quadrant.*/
	nFR = oFR; nFU = oFU;
	/*Test to find the New quadrant that the FINAL coord is in.*/
	if ((oFR==1) && (oFU==1)) {
		if ((roboA>=180) && (roboA<360) && (lNew > lOrg)) { nFR = 0;}
		if ((roboA>=90) && (roboA<270) && (hNew > hOrg))  { nFU = 0;}
	}
	else if ((oFR==1) && (oFU==0)) {
		if ((roboA>=180) && (roboA<360) && (lNew > lOrg)) { nFR = 0;}
		if ( (((roboA>=270)&&(roboA<360))||((roboA>=0)&&(roboA<90))) && (hNew > hOrg)) { nFU =
1;}
	}
	else if ((oFR==0) && (oFU==0)) {
		if ((roboA>=0) && (roboA<180) && (lNew > lOrg))   { nFR = 1;}
		if ( (((roboA>=270)&&(roboA<360))||((roboA>=0)&&(roboA<90))) && (hNew > hOrg)) { nFU =
1;}
	}
	else if ((oFR==0) && (oFU==1)) {
		if ((roboA>=0) && (roboA<180) && (lNew > lOrg))   { nFR = 1;}
		if ((roboA>=90) && (roboA<270) && (hNew > hOrg))  { nFU = 0;}
	}

	/*Calc new enviA based on which quadrant that fZ is measured in. (fZ = atan(hFin/lFin).)*/
	if (roboA==enviA) { enviA = roboA;} /*Then angle doesn't change.*/
	else if ((nFR==1)&&(nFU==1)) { enviA = 90 - fZ;}
	else if ((nFR==1)&&(nFU==0)) { enviA = 90 + fZ;}
	else if ((nFR==0)&&(nFU==0)) { enviA = 270 - fZ;}
	else if ((nFR==0)&&(nFU==1)) { enviA = 270 + fZ;}

	/*x.5 and less, rounds down. Over this rounds up.
```

```
        The rounding is neccessary because casting rounds down no matter what.*/
        rtnOb.d = ((int)(2*enviD)) - ((int)enviD);
        rtnOb.a = ((int)(2*enviA)) - ((int)enviA);

        return rtnOb;
}
int testDist(coord_t thisOb, coord_t thatOb) {

        /*(enviDist, enviAng) are the coordinates of the Robot*/
        double roboEA, roboED, enviD, enviA, relD = 0;
        double oZ, nZ, fZ, lOrg, hOrg, lNew, hNew, lFin, hFin, fHyp, obD, obA = 0;
        int relDist = 0;

        /*Variables for the C code calculations (convert back to "int"s at the end)*/
        roboED = (double) thisOb.d;
        roboEA = (double) thisOb.a;
        obD = (double) thatOb.d;
        obA = (double) thatOb.a;

        /*Original Robot Triangle with Origin (Org)*/
        oZ = ((int)roboEA)%90;
        if ((roboEA == 90) || (roboEA ==270)) {lOrg = roboED; hOrg = 0; }
        else if ((roboEA == 0) || (roboEA == 180)) {lOrg = 0; hOrg = roboED; }
        else {
                if ((roboEA <=90) || ((roboEA >=180)&&(roboEA<=270))) {
                        lOrg = (sin(oZ*DEG))*roboED;
                        hOrg = (cos(oZ*DEG))*roboED;
                } else {
                        hOrg = (sin(oZ*DEG))*roboED;
                        lOrg = (cos(oZ*DEG))*roboED;
                }
        }

        /*New Obstacle Triangle with Origin (New)*/
        nZ = ((int)obA)%90;
        if ((obA == 90) || (obA ==270)) {lNew = obD; hNew = 0; }
        else if ((obA == 0) || (obA == 180)) {lNew = 0; hNew = obD; }
        else {
                if ((obA<90) || ((obA>180)&&(obA<270))) {
                        lNew = (sin(nZ*DEG))*obD;
                        hNew = (cos(nZ*DEG))*obD;
                } else {
                        hNew = (sin(nZ*DEG))*obD;
                        lNew = (cos(nZ*DEG))*obD;
                }
        }

        /*Final Distance Triangle between Robot and Obstacle (Final)*/
        if ((roboEA<180)&&(obA>180) || (roboEA>180)&&(obA<180)) { lFin = fabs(lOrg + lNew);}
        else {  lFin = lOrg - lNew;}

        if ( (((roboEA<90)||(roboEA>270))&&((obA>90)&&(obA<270))) ||
        (((roboEA>90)&&(roboEA<270))&&((obA<90)||(obA>270))) ) {
                hFin = fabs(hOrg + hNew);
        } else { hFin = hOrg - hNew; }

        /*The relDistance between the two coordinates*/
        relD = sqrt((lFin*lFin)+(hFin*hFin));
        relDist = ((int)(2*relD)) - ((int)relD);

        return relDist;
}
```

D.2 Gnuplot shape generation C code

```
#include <math.h>
#include <stdio.h>
#include "obGenFunctions1.h"

int main() {
        FILE *fp;
        fp = fopen("polarObEnviObCoords.dat", "w");

        /*Init Ob values*/
        int enviDist = 100;
        int enviAng = 175;
        int j = 0;
        int first = 1;

        int debug = 0;

        coord_t thisObPoint;
        coord_t prevObPoint;
        coord_t centreOfOb;
        centreOfOb = setCoord(enviDist, enviAng);
        prevObPoint = setCoord(0,0);
        thisObPoint = setCoord(0,0);

                for (j=0; j<360; j++) {

                        thisObPoint = genOb(centreOfOb, j, 10);

                        /*Angle comes first with GNUPLOT.*/
                        if ( (first == 1) || ( (prevObPoint.d != thisObPoint.d) ||
                        (prevObPoint.a != thisObPoint.a) )  ) {
                                fprintf(fp, "%d %d\n", thisObPoint.a, thisObPoint.d);
                                prevObPoint = setCoord(thisObPoint.d,thisObPoint.a);
                                first = 0;
                        }
                }
        fclose(fp);
        return 0;
}
```

D.3 Gnuplot line generation C code

```
#include <math.h>
#include <stdio.h>
#include "obGenFunctions.h"

int main() {

        FILE *fp;
        fp = fopen("polarRoboSensor.dat", "w");

        /*Init Ob values*/
        int enviDist = 160;
        int enviAng = 210;
        int roboFacingAng = 90;
        int aRAng = 0;
        int j = 0;
        int first = 1;

        int debug = 0;

        coord_t thisObPoint;
        coord_t prevObPoint;
        coord_t centreOfOb;
        centreOfOb = setCoord(enviDist, enviAng);
        prevObPoint = setCoord(0,0);
        thisObPoint = setCoord(0,0);

                aRAng = ((roboFacingAng +360) -30)%360;

                for (j=1; j<61; j++) {

                        thisObPoint = genOb(centreOfOb, aRAng, (20+j));

                        /*Angle comes first with GNUPLOT.*/
                        if ( (first == 1) || ( (prevObPoint.d != thisObPoint.d) ||
                        (prevObPoint.a != thisObPoint.a) )  ) {
                                fprintf(fp, "%d %d\n", thisObPoint.a, thisObPoint.d);
                                prevObPoint = setCoord(thisObPoint.d,thisObPoint.a);
                                first = 0;
                        }
                }
        fclose(fp);
        return 0;
}
```

D.4 Gnuplot drawing script

```
reset

#GNUPLOT setup
set term postscript eps enhanced
set polar
set grid polar
set angles degrees
unset key
set size square
set xrange [-200:200]
set yrange [-200:200]
set output "wrapExample.eps"
set xtics 100
set ytics 100

set size 1,1
set multiplot

#ENVIRONMENT EXAMPLE PLOT
set size 0.5,0.5
set origin 0.0,0.5
set title "EXAMPLE: small"

plot "./autoGeneratedObstacleAndRobotINPUT1.dat" with lines lw 3 lt 1 lc rgb "black",
     "./autoGeneratedSensorINPUT1.dat" with lines lw 3 lt 1 lc rgb "blue",
     "./autoGeneratedSensorINPUT2.dat" with lines lw 3 lt 1 lc rgb "blue"

unset multiplot
```

D.5 Obstacle auto-generation C code

```
#include <math.h>
#include <stdio.h>
#define PI 3.14159265
#define DEG 0.01745329
#define OBMINDIST 155

int main() {
        FILE *fp;
        fp = fopen("ValidEnviObCoords.txt", "w");
        int debug = 1;

        /*INPUTS*/
        /*(enviD, enviA) are the coordinates of the Robot*/
        double roboA, enviD, enviA = 0;
        double oZ, nZ, fZ, lOrg, hOrg, lNew, hNew, lFin, hFin = 0;
        int furtherRight, furtherUp, oFR, oFU, nFR, nFU = 0;

        /*Variables from Promela ("now." prefix required)*/
        int roboAng = 142; /*Angle ob is to previous one*/
        int enviDist = 199;
        int enviAng = 96;
        int i = 0;
        /*Variables for the C code calculations (convert back to "int"s at the end)*/
        roboA = (double) roboAng;
        enviD = (double) enviDist;
        enviA = (double) enviAng;

        /*First line of obstacles.*/
        fprintf(fp, "arrObs[%d].d = %d; arrObs[%d].a = %d; \n", i, enviDist, i, enviAng);

        for (i=1; i<20; i++) {

                /*Triangle Original*/
                oZ = ((int)enviA)%90;
                if ((enviA == 90) || (enviA ==270)) {lOrg = enviD; hOrg = 0; }
                else if ((enviA == 0) || (enviA == 180)) {lOrg = 0; hOrg = enviD; }
                else {
                        if ((enviA <=90) || ((enviA >=180)&&(enviA<=270))) {
                                lOrg = (sin(oZ*DEG))*enviD;
                                hOrg = (cos(oZ*DEG))*enviD;
                        } else {
                                hOrg = (sin(oZ*DEG))*enviD;
                                lOrg = (cos(oZ*DEG))*enviD;
                        }
                }

                /*Triangle New*/
                /*Changed around h and l as I suspect they're wrong. CHECK CONVERSION FUNCTION.*/
                nZ = ((int)roboA)%90;
                if ((roboA == 90) || (roboA ==270)) {lNew = OBMINDIST; hNew = 0; }
                else if ((roboA == 0) || (roboA == 180)) {lNew = 0; hNew = OBMINDIST; }
                else {
                        if ((roboA<90) || ((roboA>180)&&(roboA<270))) {
                                lNew = (sin(nZ*DEG))*OBMINDIST;
                                hNew = (cos(nZ*DEG))*OBMINDIST;
                        } else {
                                hNew = (sin(nZ*DEG))*OBMINDIST;
                                lNew = (cos(nZ*DEG))*OBMINDIST;
                        }
                }

                /*Triangle Final*/
                if ((enviA<180)&&(roboA>180) || (enviA>180)&&(roboA<180)) { lFin = fabs(lOrg - lNew);}
                else {  lFin = lOrg + lNew;}

                if ( (((enviA<90)||(enviA>270))&&((roboA>90)&&(roboA<270))) ||
                (((enviA>90)&&(enviA<270))&&((roboA<90)||(roboA>270))) ) {
                        hFin = fabs(hOrg - hNew);
                } else { hFin = hOrg + hNew; }

                /**/
                if ((hFin!=0)&&(lFin!=0)) { fZ = (atan(hFin/lFin)*(180/PI)); }
                else { fZ = 0;}
```

```
                /*The new enviDist for the robot's new position*/
                enviD = sqrt((lFin*lFin)+(hFin*hFin));

                /*Set up original Quadrant*/
                if ((enviA >=0) && (enviA <90))          { oFR = 1; oFU = 1;}
                else if ((enviA >= 90) && (enviA <180))  { oFR = 1; oFU = 0;}
                else if ((enviA >= 180) && (enviA <270)) { oFR = 0; oFU = 0;}
                else if ((enviA >= 270) && (enviA <360)) { oFR = 0; oFU = 1;}

                /*Assign default values incare we do not leave the current quadrant.*/
                nFR = oFR; nFU = oFU;
                /*Test to find the New quadrant that the FINAL coord is in.*/
                if ((oFR==1) && (oFU==1)) {
                        if ((roboA>=180) && (roboA<360) && (lNew > lOrg)) { nFR = 0;}
                        if ((roboA>=90) && (roboA<270) && (hNew > hOrg))  { nFU = 0;}
                }
                else if ((oFR==1) && (oFU==0)) {
                        if ((roboA>=180) && (roboA<360) && (lNew > lOrg)) { nFR = 0;}
                        if ( (((roboA>=270)&&(roboA<360))||((roboA>=0)&&(roboA<90))) && (hNew > hOrg))
{ nFU = 1;}
                }
                else if ((oFR==0) && (oFU==0)) {
                        if ((roboA>=0) && (roboA<180) && (lNew > lOrg))   { nFR = 1;}
                        if ( (((roboA>=270)&&(roboA<360))||((roboA>=0)&&(roboA<90))) && (hNew > hOrg))
{ nFU = 1;}
                }
                else if ((oFR==0) && (oFU==1)) {
                        if ((roboA>=0) && (roboA<180) && (lNew > lOrg))   { nFR = 1;}
                        if ((roboA>=90) && (roboA<270) && (hNew > hOrg))  { nFU = 0;}
                }

                /*Calc new enviA based on which quadrant that fZ is measured in. (fZ = atan(hFin/
lFin).)*/
                if (roboA==enviA) { enviA = roboA;} /*Then angle doesn't change.*/
                else if ((nFR==1)&&(nFU==1))      { enviA = 90 - fZ;}
                else if ((nFR==1)&&(nFU==0)) { enviA = 90 + fZ;}
                else if ((nFR==0)&&(nFU==0)) { enviA = 270 - fZ;}
                else if ((nFR==0)&&(nFU==1)) { enviA = 270 + fZ;}

                /*x.5 and less, rounds down. Over this rounds up.
                The rounding is neccessary because casting rounds down no matter what.*/
                enviDist = ((int)(2*enviD)) - ((int)enviD);
                enviAng = ((int)(2*enviA)) - ((int)enviA);

                /*arrObs[0].d=80; arrObs[0].a=90;*/
                fprintf(fp, "arrObs[%d].d = %d; arrObs[%d].a = %d; \n", i, enviDist, i, enviAng);

                /*Reset to the new coords of the latest obstacle.*/
                roboA = roboA + 91;
                enviD = enviDist;
                enviA = enviAng;
        }
        fclose(fp);
        return 0;
}
```

Bibliography

[1] G. Holzmann, *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[2] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker, "PRISM: A Tool for Automatic Verification of Probabilistic Systems," *LNCS*, vol. 3920/2006, pp. 441–444, 2006.

[3] R. Kirwan, A. Miller, B. Porr, and P. Di Prodi, "Formal modeling of robot behavior with learning," *Neural Computation*, vol. 25, no. 11, pp. 2976–3019, 2013.

[4] A. Miller, R. Kirwan, B. Porr, and P. Di Prodi, "Model Checking for Improved Adaptive Behaviour," in *Proceedings of the IET conference on Control and Automation*, June 2013.

[5] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach 2nd Edition*. Pearson Education International, 2003.

[6] R. Sutton and A. Barto, "Reinforcement learning: An introduction," *MIT Press*, 1996.

[7] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. Cambridge, MA: The MIT Press, 1999.

[8] V. Braitenberg, *Vehicles: Experiments in Synthetic Psychology*. Colorado: Bradford Book, 1984.

[9] W. Walter, *The Living Brain*. London: G. Duckworth, 1953.

[10] B. Porr and F. Wörgötter, "Strongly improved stability and faster convergence of temporal sequence learning by utilising input correlations only," *Neural Computation*, vol. 18, no. 6, pp. 1380–1412, 2006.

[11] A. Pnueli, "The temporal logic of programs," *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pp. 46–57, November 1977.

[12] H. Hansson and B. Jonsson, "A Logic for Reasoning about Time and Reliability," *Formal Aspects of Computing*, vol. 6, no. 5, pp. 512–535, September 1994.

[13] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of Probabilistic Real-time Systems," in *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, ser. LNCS, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 585–591.

[14] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems*, 2nd ed. Cambridge University Press, 2004.

[15] S. Meyn and R. Tweedie, *Markov Chains and Stochastic Stability*, 2nd ed. Springer-Verlag, 2005.

[16] M. Kwiatkowska, G. Norman, and D. Parker, *Modelling and Verification of Probabilistic Systems*, ser. CRM. American Mathematical Society, 2004, vol. 23.

[17] W. Jamroga, "A Temporal Logic for Markov Chains," *International Conference on Autonomous Agents*, vol. 2, pp. 697–704, 2008.

[18] S. Akers, "Binary Decision Diagrams," *IEEE Transactions on Computers*, vol. c-27, pp. 509–516, 1978.

[19] E. Clarke, M. Fujita, P. McGeer, K. McMillan, J. Yang, and X. Zhao, "Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix

Representation." http://repository.cmu.edu/compsci/453, 1993, p. Paper 453.

[20] A. Miller, A. Donaldson, and M. Calder, "Symmetry in temporal logic model checking," *ACM Computing Surveys (CSUR)*, vol. 38, no. 8, 2006.

[21] E. Emerson and J. Halpern, ""Sometimes" and "not never" revisited: on branching versus linear time temporal logic," *ACM*, vol. 3382, no. 1, pp. 151–178, January 1986.

[22] E. Emerson and E. Clarke, "Using branching time logic to synthesize synchronization skeletons," *Science of Computer Programming*, vol. 2, no. 3, pp. 241–266, December 1982.

[23] A. Miller and A. Donaldson, "Property preservation in Quotient Structures," University of Glasgow, Tech. Rep. TR-2008-270, 2008.

[24] M. Kwiatkowska, G. Norman, and D. Parker, "Stochastic Model Checking," *LNCS*, pp. 220–270, 2007.

[25] M. Vardi and P. Wolper, "An automata-theoretic approach to automatic program verification (preliminary report)," in *Proceedings of the 1st Annual IEEE Symposium on Logic in Computer Science*. Cambridge, MA, USA: IEEE Computer Society Press, June 1986, pp. 332–344.

[26] P. Wolper, M. Vardi, and A. Sistla, "Reasoning about Infinite Computation Paths," in *Proceedings of the 4th IEEE Symposium on Foundations of Computer Science*. Tucson, AZ, USA: IEEE Computer Society, 1983, pp. 185–194.

[27] M. Vardi and P. Wolper, "Reasoning about Infinite Computations," *Information and Computation*, vol. 115, pp. 1–37, 1994.

[28] D. Peled, "Ten years of partial order reduction," *LNCS*, vol. 1427, pp. 17–28, 1998.

[29] A. Mazurkiewicz, "Trace theory, Advances in Petri Nets," *LNCS*, vol. 255, pp. 279–324, 1986.

[30] A. Donaldson, "Automatic Techniques for Detecting and Exploiting Symmetry in Model Checking," Ph.D. dissertation, University of Glasgow, June 2007.

[31] A. Donaldson and A. Miller, "Extending Symetry Reduction Techniques to a realistic model of Computation," *Electronic Notes in Theoretical Computer Science*, vol. 185, pp. 63–76, 2007.

[32] T. Henzinger, P. Ho, and H. Wong-Toi, "HYTECH: a model checker for hybrid systems," *International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1-2, pp. 110–122, 1997.

[33] U. Furbach, J. Murray, F. Schmidsberger, and F. Stolzenburg, "Hybrid Multiagent Systems with Timed Synchronization–Specification and Model Checking," *LNCS*, vol. 4908/2008, pp. 205–220, 2008.

[34] T. Henzinger, P. Ho, and H. Wong-Toi, "A User Guide to HyTech," *Technical Report*, vol. TR95-1532, 1995.

[35] T. Henzinger and H. Wong-Toi, "Using HYTECH to Synthesize Control Parameters for a Steam Boiler," *LNCS*, vol. 1165/1996, pp. 265–282, 1996.

[36] G. Frehse, "PHAVer: algorithmic verification of hybrid systems past HyTech," *International Journal on Software Tools for Technology Transfer*, vol. 10, no. 3, pp. 263–279, 2008.

[37] B. Silva, K. Richeson, B. Krogh, and A. Chutinan, "Modeling and Verifying Hybrid Dynamic Systems Using CheckMate," *Automation of Mixed Processes*, 2000.

[38] S. Ratschan and Z. She, "Safety verification of hybrid systems by constraint propagation-based abstraction refine- ment," *ACM Trans. Embedded Comput. Syst.*, vol. 6, no. 1, 2007.

[39] C. Herde, A. Eggers, M. Franzle, and T. Teige, "Analysis of Hybrid Systems using HySAT," *ICONS*, vol. Proceedings of the Third International Conference on Systems, pp. 196–201, 2008.

[40] C. Power and A. Miller, "Prism2Promela," *Qualitative Evaluation of Systems, IEEE*, pp. 79–80, September 2008.

[41] S. Shoham and O. Grumberg, "3-Valued abstraction: More precision at less cost," *Information and Computation*, vol. 206, pp. 399–410, 2006.

[42] O. Grumberg, "2-Valued and 3-Valued Abstraction-Refinement in Model Checking," in *Logics and Languages for Reliability and Security*, ser. NATO Science for Peace and Security Series - D: Information and Communication Security. IOS Press, 2010, vol. 25, pp. 105–128.

[43] Z. Andraus, M. Liffiton, and K. Sakallah, "CEGAR-Based Formal Hardware Verification: A Case Study," *Technical Report CSE-TR-531-07, University of Michigan*, 2007.

[44] R. Dearden and C. Boutilier, "Abstraction and Approximate Decision Theoretic Planning," *Artificial Intelligence*, vol. 89, pp. 219–283, 1997.

[45] E. Clarke, O. Grumberg, and D. Long, "Model Checking and Abstraction," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 5, pp. 1512–1542, September 1994.

[46] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-Guided Abstraction Refinement for Symbolic Model Checking," *Journal of the ACM*, vol. 50, no. 5, pp. 752–794, 2003.

[47] M. Wooldridge, *An Introduction to MultiAgent Systems*. John Wiley and Sons, 2009, vol. 2.

[48] M. Dastani, "2APL: a practical agent programming language," *Autonomous Agents and Multi-Agent Systems*, vol. 16, no. 3, pp. 214–248, 2008.

[49] K. Hindriks, F. De Boer, W. Van Der Hoek, and J. Meyer, "Agent Programming in 3APL," in *Autonomous Agents and Multi-Agent Systems*. Springer, 1999, vol. 2(4), pp. 357–401.

[50] A. Rao, "AgentSpeak(L): BDI agents speak out in a logical computable language," in *The 7th European Workshop on Modelling Autonomous Agents in a Multi-agent world (MAAMAW'96)*, ser. Lecture notes in Artificial Intelligence. Eindhoven, The Netherlands: Springer, 1996, pp. 42–55.

[51] R. Bordini, J. Hübner, and M. Wooldridge, *Programming Multi-Agent Systems in AgentSpeak Using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons, 2007.

[52] M. Fisher, "Temporal Development Methods for Agent-Based Systems," in *Autonomous Agents and Multi-Agent Systems*, J. Rosenschein and P. Stone, Eds., vol. 10(1). Springer, 2005, pp. 41–66.

[53] P. Kouvaros and A. Lomuscio, "Automatic verification of parameterised multi-agent systems," in *AAMAS*, 2013, pp. 861–868.

[54] A. Lomuscio, H. Qu, and F. Raimondi, "MCMAS: A Model Checker for the Verification of Multi-Agent Systems," in *CAV*. Springer, 2009, pp. 682–688.

[55] L. Dennis, M. Fisher, M. Webster, and R. Bordini, "Model Checking agent programming languages," *Automated Software Engineering*, vol. 19, no. 1, pp. 5–63, 2012.

[56] L. Dennis, M. Fisher, and M. Webster, "Using Agent JPF to Build Models for Other Model Checkers," in *Computational Logic in Multi-Agent Systems - 14th International Workshop, CLIMA XIV, Corunna, Spain, September 16-18, 2013. Proceedings*, ser. Lecture Notes in Computer Science, vol. 8143. Springer, 2013, pp. 273–289.

[57] R. Bordini, M. Fisher, W. Visser, and M. Wooldridge, "Verifying Multi-agent Programs by Model Checking," *Autonomous Agents and Multi-Agent Systems*, vol. 12, no. 2, pp. 239–256, 2006.

[58] W. Visser, K. Havelund, G. Brat, and S. Park, "Model Checking Programs," in *ASE*, 2000, pp. 3–12.

[59] K. Vikhorev, N. Alechina, and B. Logan, "Agent programming with priorities and deadlines," in *Proceedings of the Tenth International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2011)*, Taipei, Taiwan, May 2011, pp. 397–404.

[60] K. Vikhorev, N. Alechina, R. Bordini, and B. Logan, "An Operational Semantics for AgentSpeak(RT) (Preliminary Report)," in *Ninth International Workshop on Declarative Agent Languages and Technologies (DALT 2011), Workshop Notes*, Taipei, Taiwan, May 2011.

[61] N. Alechina, R. Bordini, J. Hubner, M. Jago, and B. Logan, "Belief Revision for AgentSpeak Agents," in *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2006)*. Hakodate, Japan: IEEE Press, May 2006, pp. 1288–1290.

[62] R. Bordini, J. Hübner, and R. Vieira, "Jason and the Golden Fleece of Agent-Oriented Programming," in *Multi-Agent Programming*. Springer, 2005, pp. 3–37.

[63] V. Da Silva and C. De Lucena, "From a Conceptual Framework for Agents and Objects to a Multi-Agent System Modeling Language," in *Autonomous Agents and Multi-Agent Systems*, vol. 9(1-2). Springer, 2004, pp. 145–189.

[64] I. Rahwan, "Guest Editorial: Argumentation in Multi-Agent Systems," *Autonomous Agents and Multi-Agent Systems*, vol. 11, no. 2, pp. 115–125, 2005.

[65] M. Wooldridge, N. Jennings, and D. Kinny, "The Gaia Methodology for Agent-Oriented Analysis and Design," *Autonomous Agents and Multi-Agent Systems*, vol. 3, no. 3, pp. 285–312, 2004.

[66] O. Pacheco and J. Carmo, "A Role Based Model for the Normative Specification of Organized Collective Agency and Agents Interaction," *Autonomous Agents and Multi-Agent Systems*, vol. 6, no. 2, pp. 145–184, 2004.

[67] A. Oluyomi, S. Karunasekera, and L. Sterling, "A comprehensive view of agent-oriented patterns," *Autonomous Agents and Multi-Agent Systems*, vol. 15, no. 3, pp. 337–377, 2007.

[68] B. Dunin-Kęplicz and R. Verbrugge, Eds., *Fundamenta Informaticae (special issue on formal aspects of multi-agent systems)*, vol. 63(2-3). IOS Press, 2004.

[69] H. Parunak and D. Weyns, "Guest editors' introduction, special issue on environments for multi-agent systems," *Autonomous Agents and Multi-Agent Systems*, vol. 14, no. 1, pp. 1–4, 2006.

[70] D. Weyns, A. Omicini, and J. Odell, "Environment as a first class abstraction in multiagent systems," *Autonomous Agents and Multi-Agent Systems*, vol. 14, no. 1, pp. 5–30, 2006.

[71] A. Helleboogh, G. Vizzari, A. Uhrmacher, and F. Michel, "Modeling dynamic environments in multi-agent simulation," *Autonomous Agents and Multi-Agent Systems*, vol. 14, no. 1, pp. 87–116, 2007.

[72] S. Whiteson, M. Taylor, and P. Stone, "Critical factors in the empirical performance of temporal difference and evolutionary methods for reinforcement learning," *Autonomous Agents and Multi-Agent Systems*, vol. 21, no. 1, pp. 1–35, 2010.

[73] L. Panait and S. Luke, "Cooperative Multi-Agent Learning: The State of the Art," *Autonomous Agents and Multi-Agent Systems*, vol. 11, no. 3, pp. 387–434, 2005.

[74] G. Chen, Z. Yang, H. He, and K. Goh, "Coordinating Multiple Agents via Reinforcement Learning," *Autonomous Agents and Multi-Agent Systems*, vol. 10, no. 3, pp. 273–328, 2005.

[75] A. Garland and R. Alterman, "Autonomous Agents that Learn to Better Coordinate," *Autonomous Agents and Multi-Agent Systems*, vol. 8, no. 3, pp. 267–301, 2004.

[76] F. Melo and M. Ribeiro, "Coordinated learning in multiagent MDPs with infinite state-space," *Autonomous Agents and Multi-Agent Systems*, vol. 21, no. 3, pp. 321–367, 2010.

[77] T. Kulvicius, C. Kolodziejski, M. Tamosiunaite, B. Porr, and F. Worgotter, "Behavioral analysis of differential hebbian learning in closed-loop systems," *Biological cybernetics*, 2010.

[78] R. Milner, "An algebraic definition of simulation between programs," in *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, 1971, pp. 481–489.

[79] G. Fainekos, H. Kress-Gazit, and G. Pappas, "Temporal Logic Motion Planning for Mobile Robots," in *ICRA*, 2005, pp. 2020–2025.

[80] R. Stocker, L. Dennis, C. Dixon, and M. Fisher, "Verifying Brahms Human-Robot Teamwork Models," in *JELIA*, 2012, pp. 385–397.

[81] M. Sierhuis, "Modeling and Simulating Work Practice. BRAHMS: a multi-agent modeling and simulation language for work system analysis and design," Ph.D. dissertation, Social Science and Informatics (SWI), 2001.

[82] S. Jeyaraman, A. Tsourdos, R. Zbikowski, and B. White, "Kripke modelling approaches of a multiple robots system with minimalist communication: A formal approach of choice," *Int. J. Systems Science*, vol. 37, no. 6, pp. 339–349, 2006.

[83] M. Kloetzer and C. Belta, "Hierarchical Abstractions for Robotic Swarms," in *ICRA*, 2006, pp. 952–957.

[84] M. Kloetzer, and C. Belta, "Temporal Logic Planning and Control of Robotic Swarms by Hierarchical Abstractions," *IEEE Transactions on Robotics*, vol. 213, no. 2, pp. 320–330, 2007. [Online]. Available: http://dx.doi.org/10.1109/TRO.2006.889492

[85] C. Dixon, A. Winfield, M. Fisher, and C. Zeng, "Towards temporal verification of swarm robotic systems," *Robotics and Autonomous Systems*, vol. 60, no. 11, pp. 1429–1441, 2012.

[86] S. Konur, C. Dixon, and M. Fisher, "Analysing robot swarm behaviour via probabilistic model checking," *Robotics and Autonomous Systems*, vol. 60, no. 2, pp. 199–213, 2012.

[87] C. Grana, "Selecting the Optimal Resolution and Conversion Frequency for A/D and D/A Converters," *Instrumentation and Measurement Technology Conference - IMTC*, pp. 1–6, May 2007.

[88] J. Troncoso, J. Sanchez, and F. Lopez, "Discretization of ISO-Learning and ICO-Learning to Be Included into Reactive Neural Networks for a Robotics Simulator," *Nature Inspired Problem-Solving Methods in Knowledge Engineering*, vol. 4528, 2007.

[89] G. Chesi and H. Yung, "Performance limitation analysis in visual servo systems: Bounding the location error introduced by image points matching," *IEEE International Conference on Robotics and Automation, 2009. ICRA '09*, pp. 695–700, 2009.

[90] K. Larsen, P. Patterson, and W. Yi, "Uppaal in a nutshell," *International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1-2, pp. 134–152, 1997.

[91] M. Newborn, *Automated Theorem Proving: Theory and Practice*. Springer, 2001.